

TU Ilmenau, Fakultät IA
Institut TI, Fachgebiet Automaten und Formale Sprachen
Dr. E. Hübel



**Formale Sprachen
und
Komplexität
Vorlesung SS2009**

Inhaltsverzeichnis

1	Einleitung	5
2	Algorithmus	6
2.1	Algorithmenbegriff	6
2.2	Beispiel Kreuztisch	7
2.2.1	Aufgabenstellung	7
2.2.2	Interpretation im Sinne der theoretischen Informatik, genauer im Sinne der Automatentheorie	8
3	Endlicher Automat	9
3.1	Verbale Beschreibung	9
3.2	Deterministischer endlicher Automat (DFA)	9
3.3	Äquivalenzbetrachtungen	10
3.3.1	Zustandsäquivalenz (\approx)	10
3.3.2	Automatenäquivalenz	10
3.4	Mealy-Moore-Äquivalenz	11
4	Reguläre Sprache, endlicher Automat	12
4.1	Grundlegende Begriffe	12
4.2	Reguläre Ausdrücke und reguläre Sprachen	14
4.3	Endlicher Automat/Akzeptor	16
4.4	Nichtdeterministischer endlicher Automat	20
4.5	Pumping Lemma	24
4.6	Abschlusseigenschaften für reguläre Sprachen	27
4.7	Entscheidbarkeitsfragen für reguläre Sprachen	28

5	Grammatiken und Chomsky-Hierarchie	29
5.1	Grammatik	29
5.2	Chomsky-Hierarchie	31
6	Kontextfreie Grammatiken und kontextfreie Sprachen	34
6.1	BACKUS-NAUR-Form	34
6.2	Chomsky-Normalform	35
6.3	Beispiele und Ableitungen	35
6.4	Ableitungsbäume, Linksableitung, Rechtsableitung	36
7	Kellerautomaten	39
7.1	Allgemeiner Kellerautomat, PDA (englisch: pushdown automaton).	39
7.2	Deterministischer Kellerautomat, DPDA	41
7.3	Beispiele	42
7.4	Top-Down-Parsing, LL-Parsing	45
7.5	Bottom-Up-Parsing, LR-Parsing	46
8	TURING-Maschinen und Berechenbarkeit	48
8.1	Algorithmus und Berechenbarkeit - eine Einführung	48
8.2	Die TURING-Maschine und -Berechenbarkeit	50
8.3	Beispiele	53
9	Aufzählbarkeit, Entscheidbarkeit, rekursive Sprachen	58
9.1	Aufzählbarkeit und Entscheidbarkeit	58
9.2	Rekursive Sprachen	61
9.3	Unentscheidbarkeit und Halteproblem	62
9.3.1	Kodierung von TURING-Maschinen	62
9.3.2	Universelle TM	63
9.3.3	Eine nicht rekursiv aufzählbare Sprache	64
9.3.4	Unentscheidbarkeit des Halteproblems	65
9.3.5	Der Satz von RICE	68

10 Komplexität - eine Einführung	71
10.1 Komplexitätsbegriff	71
10.2 Komplexitätsklassen	73
10.2.1 Komplexitätsklassen	73
10.2.2 Beispiele für Sprachen aus P und Funktionen aus FP .	74
10.3 NP-Probleme	75
10.4 Polynomialzeitreduktion und NP-Vollständigkeit	79
10.5 Der Satz von Cook	82
10.6 NP-vollständige Probleme	84
11 Literatur	88

Kapitel 1

Einleitung

Die Begriffe formale Sprachen und Komplexität stehen für Gebiete der theoretischen Informatik.

Die theoretische Informatik beschäftigt sich mit grundlegenden Ideen und Modellen, die Berechnungen zugrunde liegen.

Es besteht ein enger Zusammenhang zur praktischen Informatik, z.B. zum Compilerbau, zur logische Programmierung u.a..

Im Bereich der Sprachen sind z.B. für den Compilerbau zwei Klassen interessant, die durch endliche Automaten und durch Kellerautomaten repräsentiert werden können.

Formale Sprachen (z.B. Programmiersprachen) können durch „Automaten“ im allgemeinsten Sinne oder durch Grammatiken beschrieben werden. Während Grammatiken Wörter der Sprache erzeugen, können Automaten bei vorgegebenem Wort feststellen, ob dieses zur Sprache gehört oder nicht (Automat = Akzeptor).

Eine spezielle Sprachklasse kann außerdem durch Ausdrücke beschrieben werden, die reguläre Ausdrücke genannt werden.

In dieser Vorlesung werden wir uns auch mit diesen Darstellungsmöglichkeiten für Sprachen befassen.

Das Ziel der Vorlesung besteht darin, Ihnen ausgewählte Grundlagen der theoretischen Informatik zu vermitteln.

Kapitel 2

Algorithmus

2.1 Algorithmenbegriff

(als zentraler Begriff der theoretischen Informatik)

Unter dem Begriff Algorithmus versteht man in der Informatik eine Vorschrift zur schrittweisen Verarbeitung von Eingabeobjekten in Ausgabeobjekte mit folgenden Eigenschaften:

- Ein Algorithmus gilt für eine **Klasse** von Objekten.
(Ein Algorithmus löst eine Klasse von Problemen.)
- Ein Algorithmus heißt **determiniert**, wenn er bei demselben Eingabewert und derselben Startbedingung stets dasselbe Ergebnis liefert (↑ Determiniertheit), bei unterschiedlichen Ergebnissen (, auch falschen) **nichtdeterminiert**.
- Ein Algorithmus heißt **deterministisch**, wenn er zu jedem Zeitpunkt seiner Ausführung höchstens eine Möglichkeit der Fortsetzung besitzt. (↑ Determinismus), ansonsten **nichtdeterministisch**.

Ein nichtdeterministischer Algorithmus kann determiniert (z.B. Quicksort) oder nichtdeterminiert sein.

Nichtdeterminismus spielt in der Komplexitätstheorie eine große Rolle. Er ist ein wichtiges Konzept z.B. bei der Untersuchung der Laufzeit von Algorithmen (↑ NP) oder bei der Beschreibung der Größe von Prozessen.

- Ein Algorithmus **terminiert**, wenn er nach endlich vielen Schritten ein Resultat liefert und anhält.
- Die Beschreibung eines Algorithmus besitzt **endliche Länge** (statische Finitheit).
- Der Algorithmus belegt bei der Abarbeitung zu jedem Zeitpunkt **endlich viel Platz** (dynamische Finitheit).

2.2 Beispiel Kreuztisch

2.2.1 Aufgabenstellung

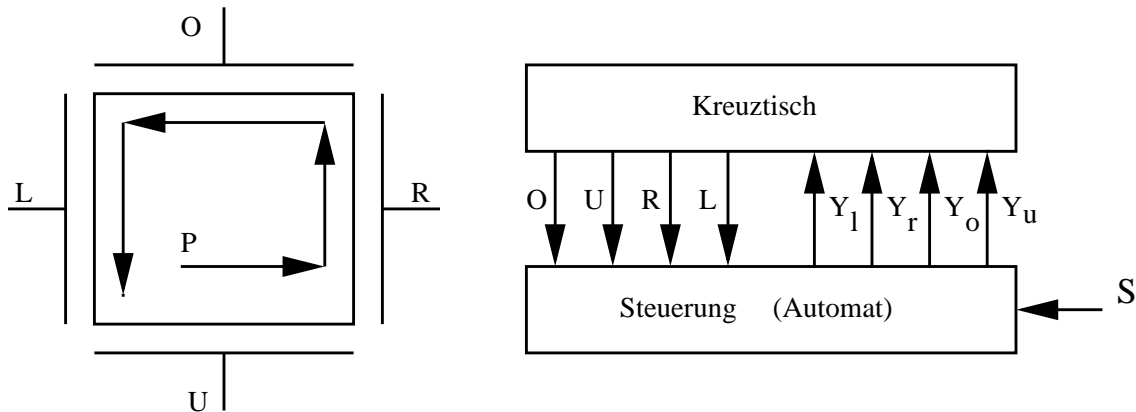


Bild 2.1: Kreuztisch

Algorithmus:

Nach Drücken der Starttaste S bewegt sich der Punkt P nach rechts.

Nach Erreichen der rechten Kante (R) bewegt sich der Punkt P nach oben.

Nach Erreichen der oberen Kante (O) bewegt sich der Punkt P nach links.

Nach Erreichen der linken Kante (L) bewegt sich der Punkt P nach unten.

Nach Erreichen der unteren Kante (U) hält der Punkt P an.

Der Vorgang kann erst erneut gestartet werden, wenn die Starttaste S nicht gedrückt ist.

⇒ (abbrechender) determinierter Algorithmus

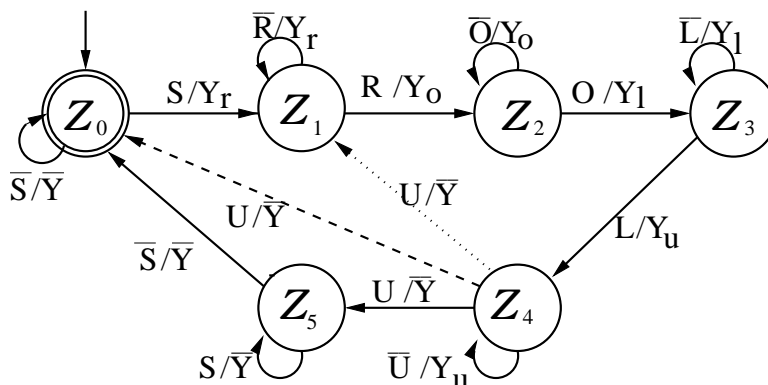


Bild 2.2: Kreuztisch-Algorithmus

- - - : Die Bedingung, dass die Starttaste S vor erneutem Start losgelassen

wird, ist entfallen. Der Algorithmus ist nicht abbrechend, falls die Starttaste S gedrückt bleibt.

. . . : Der Algorithmus ist nicht abbrechend.

⇒ im praktischen Sinne ist beides unerwünscht

2.2.2 Interpretation im Sinne der theoretischen Informatik, genauer im Sinne der Automatentheorie

Ich interpretiere die Belegungen als **Buchstaben** eines Alphabetes:

- Eingabealphabet $\Sigma = \{\bar{S}, S, \bar{R}, R, \bar{L}, L, \bar{U}, U, \bar{O}, O\}$
z.B. kann U (unten) bedeuten: unten links, unten rechts usw.
- Ausgabealphabet $\Gamma = \{Y_r, Y_l, Y_u, Y_o, \bar{Y}\}$
- Zustandsalphabet $Q = \{Z_0, Z_1, Z_2, Z_3, Z_4, Z_5\}$
- dazu kommt eine Überföhrungsfunktion $\delta : Q \times \Sigma \rightarrow Q$, die angibt, welchen Zustand $Z_i \in Q$ die Steuerung (der Automat) einnimmt, wenn sie sich in einem Zustand $Z_j \in Q$ befunden hat und in diesem eine Eingabe $x \in \Sigma$ erfolgte.
- dazu kommt eine Ausgabefunktion λ , die angibt, welche Ausgabe $y \in \Gamma$ der Automat liefert $\lambda : Q \times \Sigma \rightarrow \Gamma$.

Wird der Algorithmus unserer Aufgabenstellung in eine Schaltung umgesetzt, erhalten wir einen im weitesten Sinne deterministischen endlichen Automaten als Gerät. (Siehe technische Informatik)

Kapitel 3

Endlicher Automat

3.1 Verbale Beschreibung

Formal ist ein endlicher Automat ein mathematisches Modell für ein Gerät, das Informationen endlicher Länge Zeichen für Zeichen einliest, das eingelesene Zeichen sofort verarbeitet (, eine Ausgabe erzeugt) und nach kompletter Eingabe der Information in einem Zustand liegenbleibt.

Die Anzahl der Zustände ist endlich.

3.2 Deterministischer endlicher Automat (DFA)

Def. 3.1: Ein **deterministischer endlicher Automat** (im Sinne der technischen Informatik) ist ein 6-Tupel $M = (\Sigma, \Gamma, Q, \delta, \lambda, q_0)$ mit

$\Sigma :$	Eingabealphabet
$\Gamma :$	Ausgabealphabet
$Q :$	Zustandsalphabet mit
$q_0 \in Q$	Initialzustand
$\delta : Q \times \Sigma \rightarrow Q$	Zustandsüberföhrungsfunktion
$\lambda : Q \times \Sigma \rightarrow \Gamma$	(Mealy-)Ausgabefunktion
Spezialfall der Ausgabefunktion:	
$\mu : Q \rightarrow \Gamma$	Moore-Ausgabefunktion mit
	$\lambda(q_i, x_k) = \mu(\delta(q_i, x_k)) = \mu(q_j) = y_l$

$M_\mu = (\Sigma, \Gamma, Q, \delta, \mu, q_0)$ heißt **Moore-Automat**, der oben definierte Automat heißt **Mealy-Automat**.

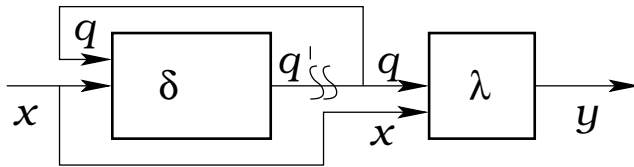


Bild 3.1: Mealy-Automat

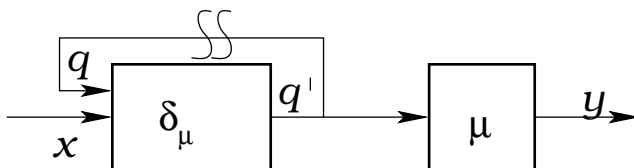


Bild 3.2: Moore-Automat

Die Unterscheidung in Moore- und Mealy-Automat ist für den Techniker interessant, für den Theoretiker weniger, da diesen die Ausgabefunktion im Allgemeinen nicht interessiert. (Siehe technische Informatik)

3.3 Äquivalenzbetrachtungen

3.3.1 Zustandsäquivalenz (\approx)

Def. 3.2 Zustände q_i, q_j sind äquivalent, wenn gilt: $q_i \approx q_j \Leftrightarrow \forall k \{ ([\delta(q_i, x_k) = \delta(q_j, x_k)] \vee [\delta(q_i, x_k) \approx \delta(q_j, x_k)]) \wedge [\lambda(q_i, x_k) = \lambda(q_j, x_k)] \}$ d.h.: gleiche Eingabefolgen liefern für q_i und q_j gleiche Ausgabefolgen.

\Rightarrow Algorithmus zur Bestimmung von Äquivalenzklassen,
Ersetzung der Klassen durch einen Repräsentanten.

\Rightarrow z.B. Verfahren nach Aufenkamp und Hohn

3.3.2 Automatenäquivalenz

Satz 3.1: Zwei Automaten M_i und M_j mit Eingabealphabeten $\Sigma_i = \Sigma_j$ und Ausgabealphabeten $\Gamma_i = \Gamma_j$ sind äquivalent, wenn zu jedem Zustand q_i von M_i mindestens ein ihm äquivalenter Zustand q_j von M_j existiert und umgekehrt.

Satz 3.2: Für alle äquivalenten Automaten gibt es einen Automaten M_{min} , auf den sich alle Automaten zustandshomomorph abbilden lassen, d.h. $M_i \approx M_j \Leftrightarrow M_{i_{min}} \text{ und } M_{j_{min}}$ sind eindeutig aufeinander abbildbar.

3.4 Mealy-Moore-Äquivalenz

Der Moore-Automat kann als Spezialfall des Mealy-Automaten betrachtet werden.

Satz 3.3: Zu jedem Mealy-Automaten M gibt es einen ihm äquivalenten Moore-Automaten M_μ :

Es existieren 2 mögliche Konstruktionsverfahren:

1. Aus der Menge $Q \times \Gamma$ des Mealy-Automaten folgt die Zustandsmenge Q_μ .
2. Aus der Menge $Q \times \Sigma$ des Mealy-Automaten folgt die Zustandsmenge Q_μ .

Das Modell des endlichen Automaten soll als Darstellungsmittel für formale Sprachen genutzt werden.

Kapitel 4

Reguläre Sprache, endlicher Automat

4.1 Grundlegende Begriffe

Grundbegriff Zeichen - wird nicht weiter definiert.

Ein Alphabet ist eine nichtleere, endliche Menge von Zeichen.

Ohne Zwischenraum aneinander gereihte Zeichen nennt man Wort (synonym: Zeichenkette oder Zeichenreihe oder String).

Sei $\Sigma = \{a, b, c\}$ ein Alphabet, so sind z.B. a , ab , $bcba$ Wörter.

Die Länge eines Wortes x , $\text{lgth}(x)$ oder $|x|$, ist die Anzahl der Zeichen, aus denen es besteht, z.B. $\text{lgth}(a) = 1$, $\text{lgth}(caba) = 4$.

Das leere Wort (es hat die Länge 0) wird mit ε bezeichnet.

Die Menge aller bildbaren Wörter (einschließlich des leeren Wortes) über einem gegebenen Alphabet Σ wird als Σ^* bezeichnet.

Def. 4.1: Die KLEENESche Hülle von Σ ist die Menge $\Sigma^* = \bigcup_{i=0}^{\infty} \Sigma^i$.

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \dots \cup \Sigma^n \cup \dots$ mit $\Sigma^n = \Sigma^{n-1}\Sigma = \{ab \mid a \in \Sigma^{n-1} \wedge b \in \Sigma\}$.

Das direkte Hintereinanderschreiben zweier Wörter $w, v \in \Sigma^*$ nennt man Verkettung (oder Konkatenation) $\otimes : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ (ε neutrales Element).

Für $\otimes(v, w) = v \otimes w$ schreiben wir auch kurz vw .

Def. 4.2: Die positive Hülle von Σ ist $\Sigma^+ = \bigcup_{i=1}^{\infty} \Sigma^i = \Sigma^* - \{\varepsilon\}$.

Seien $\Sigma_1 = \{0, 1\}$ und $\Sigma_2 = \{a, b, c, d\}$ zwei Alphabete.

Dann ist

$\Sigma_1^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, \dots\}$,
 $\Sigma_2^* = \{\varepsilon, a, b, c, d, aa, ab, ac, ad, ba, bb, bc, bd, ca, cb, cc, cd, da, \dots\}$.

Def. 4.3: Die n -te **Potenz** eines Wortes $x \in \Sigma^*$ ist eine Abbildung der Form $Pot : \Sigma^* \times \mathbb{N} \rightarrow \Sigma^*$, die wie folgt definiert wird:

- (i) $Pot(x, 0) = x^0 = \varepsilon$;
- (ii) $Pot(x, n) = x^n = x^{n-1}x$.

Eine **formale Sprache** L ist nun zunächst eine Teilmenge aller Wörter über einem vorgegebenen Alphabet $\Sigma : L \subseteq \Sigma^*$.

Bem.: Die Sprachen \emptyset , $\{\varepsilon\}$ und Σ^* sind trivial und für praktische Zwecke uninteressant. I. Allg. werden uns nur echte Teilmengen interessieren. Somit ist $\mathcal{P}(\Sigma^*)$ die Menge aller formalen Sprachen über dem Alphabet Σ .

Wir erweitern die bekannten Operationen auf Sprachen wie folgt:

Verkettung $\otimes : \mathcal{P}(\Sigma^*) \times \mathcal{P}(\Sigma^*) \rightarrow \mathcal{P}(\Sigma^*)$.

Es sei Σ ein Alphabet und es seien $L_1, L_2 \subseteq \Sigma^*$ Sprachen.

Dann ist: $\otimes(L_1, L_2) = L_1 \otimes L_2 = L_1 L_2 = \{x y \mid x \in L_1 \wedge y \in L_2\}$,

d.h. $L_1 L_2$ ist die Menge aller möglichen Verkettungen jeweils eines Wortes x aus L_1 mit einem Wort y aus L_2 .

Potenz $Pot : \mathcal{P}(\Sigma^*) \times \mathbb{N} \rightarrow \mathcal{P}(\Sigma^*)$.

Sei Σ ein Alphabet und $L \subseteq \Sigma^*$ ein Sprache. Dann ist

- (i) $Pot(L, 0) = L^0 = \{\varepsilon\}$;
- (ii) $Pot(L, n) = L^n = L^{n-1}L$.

Die **Kleenesche Hülle** von L ist die Menge $L^* = \bigcup_{i=0}^{\infty} L^i$.

Die **positive Hülle** von L ist die Menge $L^+ = \bigcup_{i=1}^{\infty} L^i = L^*L$.

Sei $L_c = \{10, 1\}$ und $L_d = \{011, 11\}$. $L_c L_d = ?$ und $L_d L_c = ?$.

Beispiel 4.1: $L_1 = \{\varepsilon, a, a a, a a a a, a^8, a^{16} \dots\} \stackrel{\wedge}{=} \text{Aufzählen aller Wörter.}$

$L_1 = \{a^n \mid n = 2^k, k \in \mathbb{N}\} \cup \{\varepsilon\}$ wird über ein Prädikat definiert.

Wenn das Prädikat berechenbar ist, dann ist die Sprache entscheidbar.

Beispiel 4.2: Rekursive Definition der Sprache L_2

- (i) $\varepsilon \in L_2, a \in L_2,$
- (ii) $x \in L_2 \Rightarrow x x \in L_2,$
- (iii) andere Elemente als die nach (i) und (ii) enthält L_2 nicht.

Diese rekursive Definition hat generativen Charakter.

Bemerkung: $L_1 = L_2.$

Hieraus ergibt sich unmittelbar die Problemstellung, eine Verbindung zwischen analysierenden und generativen Sprachdefinitionen herzustellen und die Äquivalenz verschiedener Sprachdefinitionen bzw. die Gleichheit zweier Sprachen festzustellen.

Dies soll nun am Beispiel einer relativ einfachen Sprachklasse erläutert werden.

4.2 Reguläre Ausdrücke und reguläre Sprachen

Die Darstellung von Sprachen ist für eine bestimmte Klasse von Sprachen mitunter einfacher über Ausdrücke anzugeben als über Mengen

\Rightarrow Reguläre Ausdrücke über $\Sigma.$

Def. 4.4: Σ sei ein Alphabet.

Reguläre Ausdrücke (rA) werden dann wie folgt definiert:

- (i) \emptyset, ε und $a \in \Sigma$ sind rA über $\Sigma.$
- (ii) Wenn r und s rA über Σ dann auch $(r + s), (r \cdot s) = (r s)$ und $(r^*).$
- (iii) Weitere reguläre Ausdrücke existieren nicht.

Bem.: $(r^+) = (r^*) \cdot (r) = (r) \cdot (r^*)$ erlaubt.

Wir wollen nun reguläre Ausdrücke zur Beschreibung von Sprachen verwenden und ordnen dazu jedem regulären Ausdruck p in eindeutiger Weise induktiv eine Sprache $L(p)$ zu:

- (i) Falls $p = \emptyset$, so ist $L(p) = \emptyset$.
 Falls $p = \varepsilon$, so ist $L(p) = \{\varepsilon\}$.
 Falls $p = a$ mit $a \in \Sigma$, so ist $L(p) = \{a\}$.
- (ii) Falls $p = (rs)$, wobei r und s rA sind, so ist $L(p) = L(r)L(s)$.
 Falls $p = (r + s)$, wobei r und s rA sind, so ist $L(p) = L(r) \cup L(s)$.
 Falls $p = (r^*)$, wobei r rA ist, so ist $L(p) = L(r)^*$.

Bem.: $L(r^+) = L^+(r) = (L(r))^+$

Die Klammern können entfallen, wenn dadurch keine Fehler entstehen.
 Zur Klammerersparnis in regulären Ausdrücken kann die Operatorpriorität genutzt werden: $*$ bindet stärker als \cdot und \cdot wiederum stärker als $+$.

Bem.: Durch reguläre Ausdrücke werden Sprachen generativ definiert.

Satz 4.1: Sprachen, die durch reguläre Ausdrücke definierbar sind, heißen reguläre Sprachen.

Jede reguläre Sprache ist offenbar eine formale Sprache. Dies gilt jedoch nicht umgekehrt.

Beispiel 4.3: Reguläre Sprachen, die in der Compilertechnik bei der Lexikanalyse eine Rolle spielen:

- Menge von Operatoren: $L_{Op} = \{ :, <, =, >, <=, >=, :=, *, **, +, -, /, \}$
 $r_{L_{Op}} = (: + < + = + > + <= + >= + := + * + ** + + - + /),$

- Menge von Bezeichnern:

Die Wörter der Sprache L_{Id} über dem Alphabet $\Sigma = \{a, \dots, z, 1, \dots, 0\}$ beginnen mit einem Buchstaben und es folgen beliebig viele Zeichen, bestehend aus Buchstaben oder Ziffern.

(In Programmiersprachen werden so Bezeichner gebildet.)

$L_{Id} = \{ a, b, c, \dots, z \} \{ a, b, c, \dots, z, 0, 1, 2, \dots, 9 \}^*$

$r_{L_{Id}} = (a + b + c + \dots + z) (a + b + c + \dots + z + 0 + 1 + 2 + \dots + 9)^*:$

- Menge von Schlüsselwörtern:

$L_{SW} = \{ AND, OR, BEGIN, END, \dots, ARRAY \}$

$r_{L_{SW}} = \{ AND + OR + BEGIN + END + \dots + ARRAY \}$

4.3 Endlicher Automat/Akzeptor

Wie kann aber festgestellt werden, ob ein gegebenes Wort Element einer bestimmten Sprache ist?

Man kann versuchen, das Wort Zeichen für Zeichen zu lesen und nach jedem Zeichen entscheiden, ob das Wort ausscheidet oder wie bei der weiteren Prüfung zu verfahren ist.

Das kann für reguläre Sprachen mit einem **endlichen Automaten** realisiert werden, der auch **(endlicher) Akzeptor** genannt wird. Die unterschiedlichen Entscheidungsmuster werden als Zustände bezeichnet.

Der endliche Automat startet in einem Anfangszustand und geht nach jedem gelesenen Zeichen in einen neuen Zustand über und landet in einem (akzeptierenden) Endzustand, falls das Wort zur Sprache gehört.

Def. 4.5: Ein **deterministischer endlicher Automat [DFA]** ist ein

Tupel $M = (Q, \Sigma, \delta, q_0, F)$ mit

Q endliche nichtleere Menge von Zuständen,

Σ Eingabealphabet,

$\delta : Q \times \Sigma \rightarrow Q$ Zustandsüberföhrungsfunktion,

$q_0 \in Q$ Initialzustand (auch Anfangs- oder Startzustand),

$F \subseteq Q$ Menge von (akzeptierenden) Endzuständen.

Ein DFA M **erkennt (akzeptiert) ein Wort** $x = a_1 \dots a_n \in L_M \subseteq \Sigma^*$,

indem er bei seinem „Lesen“ eine Folge von Zuständen $q_0 q_1 q_2 \dots q_n$ durchläuft, deren erster (q_0) der Startzustand und deren letzter (q_n) ein (akzeptierender) Endzustand ist.

Dabei muss gelten: $\forall i \in \{1, 2, \dots, n\} : \delta(q_{i-1}, a_i) = q_i$.

Die von einem endlichen Automaten M akzeptierte Sprache $L_M \subseteq \Sigma^*$ ist genau dann die Menge aller durch den Automaten akzeptierten Wörter.

Die **Zustandsüberföhrungsfunktion** δ wird wie folgt für Wörter erweitert: $\delta : Q \times \Sigma^* \rightarrow Q$, dann gilt

$M = (Q, \Sigma, \delta, q_0, F)$ **akzeptiert** $x \Leftrightarrow \delta(q_0, x) \in F$

$\Rightarrow L_M = \{ x \mid \delta(q_0, x) \in F \}$.

Beispiel 4.4: L_{Op} und L_{Id} seien die in Beispiel 4.3 gegebenen Sprachen:

$$L_{Op} = \{ :, <, =, >, <=, >=, :=, *, **, +, -, /, \}$$

$$L_{Id} = \{ a, b, c, \dots, z \} \{ a, b, c, \dots, z, 0, 1, 2, \dots, 9 \}^*$$

Dann könnten diese regulären Sprachen von folgenden endlichen Automaten, die bei der Lexikanalyse genutzt werden, akzeptiert werden:

$$M_{L_{Op}} = (\{q_0, q_1, q_2, q_3, q_4\}, \{ :, <, =, >, *, +, -, / \}, \delta_{Op}, q_0, \{q_1, q_2, q_4\}),$$

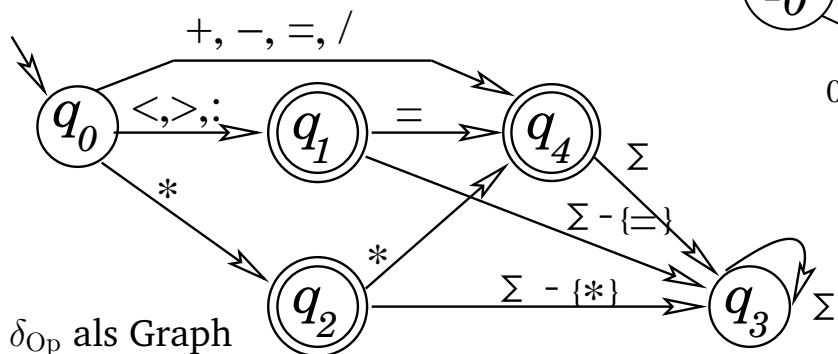
$$M_{L_{Id}} = (\{q_0, q_1\}, \{ a, b, c, \dots, z, 0, 1, 2, \dots, 9 \}, \delta_{Id}, q_0, \{q_1\}),$$

δ_{Op} Zustand	Eingabe								
	:	<	=	>	*	+	-	/	
q_0	q_1	q_1	q_4	q_1	q_2	q_4	q_4	q_4	
q_1	q_3	q_3	q_4	q_3	q_3	q_3	q_3	q_3	q_3
q_2	q_3	q_3	q_4	q_3	q_3	q_3	q_3	q_3	q_3
q_3	q_3	q_3	q_3	q_3	q_3	q_3	q_3	q_3	q_3
q_4	q_3	q_3	q_3	q_3	q_3	q_3	q_3	q_3	q_3

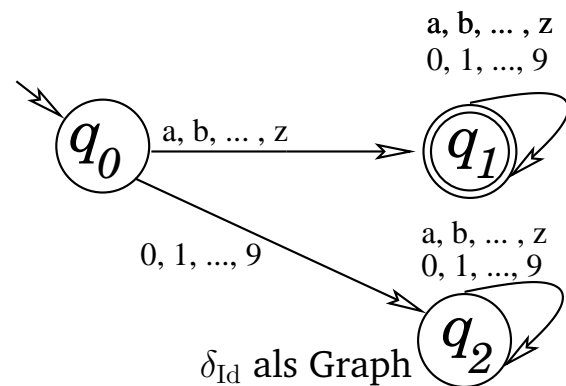
δ_{Op} als Tabelle

δ_{Id} Zustand	Eingabe	
	B	Z
q_0	q_1	q_2
q_1	q_1	q_1
q_2	q_2	q_2

δ_{Id} als Tabelle



δ_{Op} als Graph



δ_{Id} als Graph

Satz 4.2: Jede durch endliche Automaten erkennbare Sprache ist eine reguläre Sprache.

Wir führen den Beweis konstruktiv: (weglassen)

Gegeben sei ein deterministischer endlicher Automat M mit n Zuständen, die beginnend mit 0 ($q_0 =$ Startzustand) durchnummeriert werden:

$$Q = \{q_0, q_1, q_2, \dots, q_{n-1}\}.$$

Wir definieren zunächst $|Q|^2$ Sprachen, die sich im Automaten M ergeben, indem man alle Wege vom Zustand q_i zum Zustand q_j erfasst:

$$L(i, j) = \{x \in \Sigma^* \mid q_j \in \delta(q_i, x)\} \text{ für } 0 \leq i, j < n.$$

Dabei werden beliebige Wege von q_i nach q_j zugelassen.

Wir definieren nun Sprachen $L(i, j, k)$ für $0 \leq i, j < n$ und $0 \leq k \leq n$ mit folgender Bedeutung:

$$L(i, j, k) = \{x \in \Sigma^* \mid \text{die Eingabekette } x \text{ überführt den Automaten vom Anfangszustand } q_i \text{ in den Zustand } q_j, \text{ wobei alle Zwischenzustände außer } q_i \text{ und } q_j \text{ selbst nur Indizes kleiner } k \text{ haben}\}$$

Es gelten folgende Eigenschaften der Mengen $L(i, j, k)$, für $0 \leq i, j < n$ und $k = 0$ bzw. $k = n$:

Für $k = 0$ (keine Zwischenzustände) sind die Sprachen endlich (und lassen sich folglich durch einen regulären Ausdruck beschreiben):

$$L(i, j, 0) = \{a \in \Sigma \mid q_j \in \delta(q_i, a)\}, \text{ mit } i \neq j; \\ = \text{Menge der Buchstaben an den Kanten } (i, j)$$

$$L(i, i, 0) = \{a \in \Sigma \mid q_i \in \delta(q_i, a)\} \cup \{\varepsilon\} \\ = \text{Menge der Buchstaben an den Kanten } (i, i) \cup \{\varepsilon\}.$$

Für $k = n$ ergibt sich die Sprache für den Automaten mit q_i Initialzustand und q_j Endzustand:

$$L(i, j) = L(i, j, n)$$

Um die regulären Ausdrücke für $L(i, j, k)$ induktiv konstruieren zu können, benötigen wir die Beziehung zwischen den Sprachen $L(., ., k)$ und $L(., ., k + 1)$:

$$L(i, j, k + 1) = L(i, j, k) \cup L(i, k, k) L(k, k, k)^* L(k, j, k).$$

Mit Hilfe obiger Aussagen konstruieren wir reguläre Ausdrücke $r_{i,j,k}$ für $L(i, j, k)$, mit $0 \leq i, j < n$ und $0 \leq k \leq n$.

$$r_{i,j,0} := a_1 + \dots + a_n + \emptyset, \text{ mit } L(i, j, 0) = \{a_1, \dots, a_n\} \subseteq \Sigma, i \neq j$$

$$r_{i,i,0} := a_1 + \dots + a_n + \varepsilon, \text{ mit } L(i, i, 0) = \{a_1, \dots, a_n\} \cup \{\varepsilon\}$$

$$r_{i,j,k} := r_{i,j,k-1} + r_{i,k-1,k-1} r_{k-1,k-1,k-1}^* r_{k-1,j,k-1} \text{ für} \\ 0 \leq i, j < n \text{ und } 0 < k \leq n$$

Aus der Konstruktion folgt (per Induktion über k), dass

$$L(i, j, k) = L(r_{i,j,k}) \text{ für } 0 \leq i, j < n \text{ und } 0 \leq k \leq n.$$

Setzen wir $r_j := r_{0,j,n}$, für $0 \leq j < n$,

dann folgt $L(0, j) = L(0, j, n) = L(r_{0,j,n}) = L(r_j)$.

Beispiel 4.5: Überführung eines Automaten M in einen regulären Ausdruck r_M

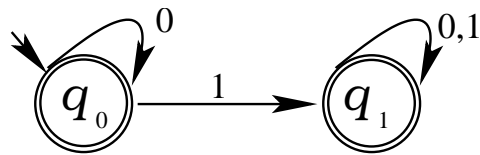


Bild 4.1: Automat M

$$r_M = r_{0,0,2} + r_{0,1,2}$$

$$r_{i,j,k} = r_{i,j,k-1} + r_{i,k-1,k-1} r_{k-1,k-1,k-1}^* r_{k-1,j,k-1}, \quad 0 \leq i, j < n \text{ und } 0 < k \leq n$$

Damit nicht alle $r_{i,j,k}$ erstellt werden, beginnen wir mit $k = 2$.

$$\begin{aligned} k = 2: \quad r_{0,0,2} &= r_{0,0,1} + r_{0,1,1} r_{1,1,1}^* r_{1,0,1} = 0^* + 0^* 1 (0 + 1 + \varepsilon)^* \emptyset = 0^* \\ r_{0,1,2} &= r_{0,1,1} + r_{0,1,1} r_{1,1,1}^* r_{1,1,1} = 0^* 1 + 0^* 1 (0 + 1 + \varepsilon)^* (0 + 1 + \varepsilon) \\ &= 0^* 1 (0 + 1 + \varepsilon)^+ = 0^* 1 (0 + 1)^* \end{aligned}$$

$$\begin{aligned} k = 1: \quad r_{0,0,1} &= r_{0,0,0} + r_{0,0,0} r_{0,0,0}^* r_{0,0,0} = 0 + \varepsilon + (0 + \varepsilon) (0 + \varepsilon)^* (0 + \varepsilon) = 0^* \\ r_{0,1,1} &= r_{0,1,0} + r_{0,0,0} r_{0,0,0}^* r_{0,1,0} = 1 + (0 + \varepsilon) (0 + \varepsilon)^* 1 = 1 + 0^* 1 = 0^* 1 \\ r_{1,0,1} &= r_{1,0,0} + r_{1,0,0} r_{0,0,0}^* r_{0,0,0} = \emptyset + \emptyset \\ r_{1,1,1} &= r_{1,1,0} + r_{1,0,0} r_{0,0,0}^* r_{0,1,0} = 0 + 1 + \varepsilon + \emptyset \end{aligned}$$

$$k = 0: \quad r_{0,0,0} = 0 + \varepsilon, \quad r_{0,1,0} = 1, \quad r_{1,0,0} = \emptyset, \quad r_{1,1,0} = 0 + 1 + \varepsilon$$

$$r_M = r_{0,0,2} + r_{0,1,2} = 0^* + 0^* 1 (0 + 1)^*$$

Satz 4.3: Zu jeder regulären Sprache L gibt es einen endlichen Automaten M_L , der diese Sprache erkennt.

Für den Beweis benötigen wir als Hilfsmittel den nichtdeterministischen endlichen Automaten.

4.4 Nichtdeterministischer endlicher Automat

Def. 4.6: Ein nichtdeterministischer endlicher Automat, kurz NFA, ist ein Tupel $(Q, \Sigma, \delta, S, F)$ mit

Q und Σ wie beim DFA,

$\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ die (nichtdet.) Zustandsüberföhrungsfunktion,

$S \subseteq Q$ eine Menge von Initialzuständen,

$F \subseteq Q$ eine Menge von Endzuständen.

In einem NFA ist es zugelassen, dass in einem Zustand q für ein Eingabezeichen $a \in \Sigma$ mehrere Folgezustände oder gar kein Folgezustand spezifiziert sind.

In diesem Fall kann der Automat in jeden beliebigen der in $\delta(q, a)$ enthaltenen Zustände übergehen.

Eine Zeichenkette gilt bereits dann als akzeptiert, wenn der Automat beim Lesen von irgendeinem der Startzustände aus eine Zustandsfolge durchlaufen kann, die auf einen Endzustand führt.

Die **Zustandsüberföhrungsfunktion** kann wieder auf die Anwendung von Wörtern erweitert werden: $\delta : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$

Die von einem NFA $N = (Q, \Sigma, \delta, S, F)$ erkannte Sprache L_N ist somit

$$L_N = \{x \in \Sigma^* \mid \delta(S, x) \cap F \neq \emptyset\}.$$

Satz 4.4: von RABIN, SCOTT: Jede von einem NFA akzeptierte Sprache ist auch durch einen DFA akzeptierbar.

Beweis erfolgt durch Konstruktion:

Sei $N = (Q, \Sigma, \delta, S, F)$ ein NFA.

Wir betrachten dazu folgenden DFA $M = (Q_M, \Sigma, \delta_M, q_{0_M}, F_M)$, wobei

$$Q_M = \mathcal{P}(Q), \quad q_M \in \mathcal{P}(Q),$$

$$\delta_M(q_M, a) = \bigcup_{q \in q_M} \delta(q, a) \text{ mit } q_M \in Q_M,$$

$$q_{0_M} = S \text{ und } F_M = \{q_M \subseteq Q \mid q_M \cap F \neq \emptyset\}.$$

Offenbar gilt jetzt für jedes $x = a_1 a_2 \dots a_n \in \Sigma^*$ mit $a_1, a_2, \dots, a_n \in \Sigma$:

$$x \in L_N \Leftrightarrow \delta(S, x) \cap F \neq \emptyset$$

$$\Leftrightarrow \exists q_{0_M}, \dots, q_{n_M} \subseteq Q \text{ mit}$$

$$(q_{0_M} = S) \wedge (\forall i \in \{1, \dots, n\} : \delta_M(q_{i-1_M}, a_i) = q_{i_M}) \wedge (q_{n_M} \cap F \neq \emptyset)$$

$$\Leftrightarrow \delta_M(q_{0_M}, x) \in F_M \Leftrightarrow x \in L_M.$$

Der konstruierte DFA muss nicht minimal sein. Insbesondere wird er viele vom Startzustand gar nicht erreichbare Zustände enthalten, die einfach entfernt werden können. Es gibt verschiedene Verfahren zur weiteren Minimierung von DFA, die hier nicht angegeben werden können.

Praktischerweise werden zur Erstellung des DFA nicht alle $q_M \in \mathcal{P}(Q)$ genutzt, sondern nur die Mengen q_M , die im NFA tatsächlich vorkommen.

Beispiel 4.6: Umwandlung NFA N in DFA M

Gegeben ist folgender NFA N mit $r_N = (0 + 1)^* + (0 + 2)^* + (0 + 3)^*$:

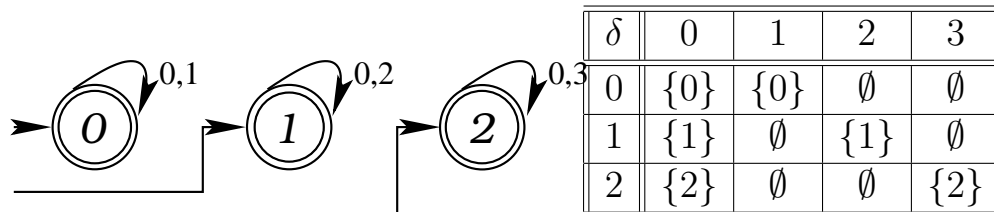


Bild 4.2: Automatengraph für δ_N des NFA und Tabelle.

$$S = \{0, 1, 2\}, F = \{0, 1, 2\}$$

DFA:

$$q_0 = \{0, 1, 2\}, F = \{\{0, 1, 2\}, \{0\}, \{1\}, \{2\}\}$$

δ	0	1	2	3
{0, 1, 2}	{0, 1, 2}	{0}	{1}	{2}
{0}	{0}	{0}	\emptyset	\emptyset
{1}	{1}	\emptyset	{1}	\emptyset
{2}	{2}	\emptyset	\emptyset	{2}
\emptyset	\emptyset	\emptyset	\emptyset	\emptyset

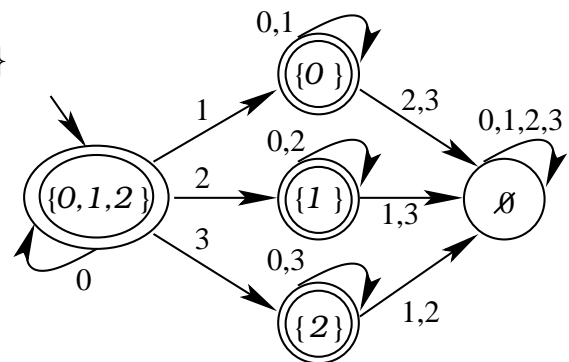


Bild 4.3: Automatengraph DFA

Nun bleibt noch zu zeigen, dass es zu jeder regulären Sprache einen NFA gibt, der diese erkennt. (siehe Satz 4.4)

Dies erfolgt induktiv über dem Aufbau regulärer Ausdrücke:

- \emptyset wird durch einen NFA N_\emptyset mit leerer Endzustandsmenge „erkannt“. Aus kompositorischen Gründen wählen wir einen Automaten mit vom Startzustand aus nicht erreichbarem Endzustand, z.B.:



Bild 4.4: Automatengraph NFA N_\emptyset

- ε wird mindestens durch folgenden NFA N_ε erkannt:

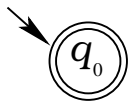


Bild 4.5: Automatengraph NFA N_ε

- Jedes a aus Σ wird offenbar erkannt durch N_a :

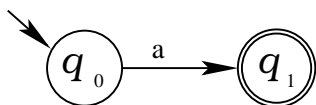


Bild 4.6: Automatengraph NFA N_a

- Es seien nun zwei reguläre Ausdrücke r_1 und r_2 gegeben, deren repräsentierte reguläre Mengen durch die NFA $N_1 = (Q_1, \Sigma, \delta_1, S_1, F_1)$ bzw. $N_2 = (Q_2, \Sigma, \delta_2, S_2, F_2)$ mit $Q_1 \cap Q_2 = \emptyset$ erkannt werden (so dass $L(r_1) = L_{N_1}$ und $L(r_2) = L_{N_2}$).

Konstruktion eines NFA $N_{r_1 r_2} = (Q, \Sigma, \delta, S, F)$ zu $r = (r_1 r_2)$ wie folgt:

$Q = Q_1 \cup Q_2$, $S = S_1$, $F = F_2$ und falls $\varepsilon \in L(r_1)$, so ist $S = S_1 \cup S_2$.

δ enthält alle Tupel von δ_1 und δ_2 , und zusätzlich ist $S_2 \subseteq \delta(q, a)$ für alle q, a mit $\delta_1(q, a) \cap F_1 \neq \emptyset$, d.h.:

$$\delta(q, a) = \begin{cases} \delta_1(q, a), & \text{falls } q \in Q_1 \wedge \delta_1(q, a) \cap F_1 = \emptyset \\ \delta_1(q, a) \cup S_2, & \text{falls } q \in Q_1 \wedge \delta_1(q, a) \cap F_1 \neq \emptyset \\ \delta_2(q, a) & \text{sonst } (q \in Q_2) \end{cases}$$

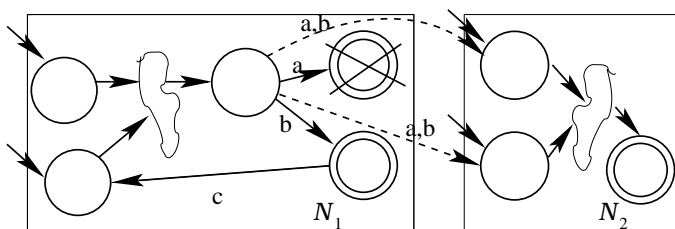


Bild 4.7: Automatengraph NFA $N_{r_1 r_2}$

Falls $\varepsilon \in L(r_1)$, dann bleiben die zuführenden Kanten der Initialzustände von N_2 erhalten, sonst nicht.

- Konstruktion eines NFA $N_{r_1+r_2} = (Q, \Sigma, \delta, S, F)$ zu $r = (r_1 + r_2)$ wie folgt:
 $Q = Q_1 \cup Q_2$, $S = S_1 \cup S_2$, $F = F_1 \cup F_2$,

δ enthält alle Tupel von δ_1 und δ_2 , d.h.: $\delta(q, a) = \begin{cases} \delta_1(q, a), & \text{falls } q \in Q_1 \\ \delta_2(q, a), & \text{falls } q \in Q_2 \end{cases}$

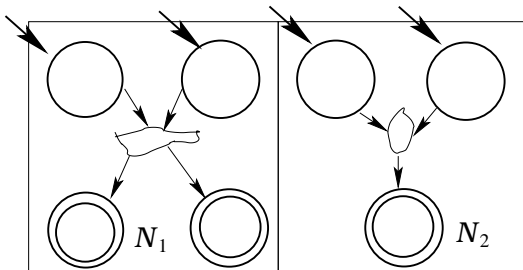


Bild 4.8: Automatengraph NFA $N_{r_1+r_2}$

- Wir konstruieren zu $r = (r_1^*)$ einen NFA $N_{r_1^*}$

$N = (Q_1 \cup Q', \Sigma, \delta, S_1 \cup Q', F_1 \cup Q')$:

(Q' muss sowohl End- als auch Initialzustand sein)

$$Q' = \begin{cases} \emptyset, & \text{falls } \varepsilon \in L(r_1) \\ \{q\} \text{ mit } q \notin Q_1 & \text{falls } \varepsilon \notin L(r_1) \end{cases}$$

δ enthält alle Tupel von δ_1 , und zusätzlich ist $S \subseteq \delta(q, a)$ für alle (q, a) mit $\delta_1(q, a) \cap F_1 \neq \emptyset$, d.h.:

$$\delta(q, a) = \begin{cases} \delta_1(q, a), & \text{falls } q \in Q_1 \wedge \delta_1(q, a) \cap F_1 = \emptyset \\ \delta_1(q, a) \cup S_1, & \text{falls } q \in Q_1 \wedge \delta_1(q, a) \cap F_1 \neq \emptyset \\ \emptyset & \text{falls } q \notin Q_1 \end{cases}$$

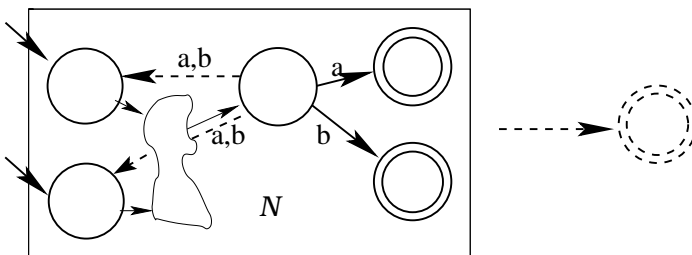


Bild 4.9: Automatengraph NFA $N_{r_1^*}$

Die konstruierten NFA akzeptieren offensichtlich die jeweiligen Sprachen.

Da die NFA in DFA und diese wiederum in reguläre Ausdrücke überführt werden können, ist nun die Äquivalenz von endlichen Automaten und regulären Ausdrücken (bzw. Mengen) bewiesen:

Satz 4.5: Die Menge der durch endliche Automaten akzeptierbaren Sprachen ist genau die Menge der regulären Sprachen.

4.5 Pumping Lemma

Satz 4.6: Pumping Lemma für reguläre Sprachen:

Sei L eine unendliche reguläre Sprache.

Dann gibt es eine Zahl $n \in \mathbb{N}$, sodass sich jedes Wort $x \in L$ mit $|x| \geq n$ als $x = uvw$ schreiben lässt mit

$|uv| \leq n$ und

$|v| \neq 0$ d.h. $v \neq \varepsilon$,

wobei für alle $i \geq 0$ gilt, dass $uv^i w \in L$.

Außerdem ist n nicht größer als die Zahl der Zustände in dem kleinsten FA, der L akzeptiert.

Beweis: (weglassen)

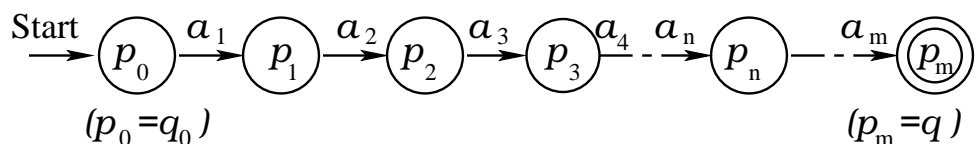
Sei L eine reguläre Sprache.

Dann ist $L = L_M$ für einen FA $M = (Q, \Sigma, \delta, q_0, F)$

n ist die Anzahl der Zustände Q , nämlich $|Q|$.

Man betrachte ein Wort $x = a_1 a_2 \dots a_m \in L$, $m \geq n$.

Es gibt eine akzeptierende Berechnung von M für x , die einem Weg P_x von q_0 nach q , mit $q \in F$ in einem Graphen G_M entspricht, dessen Kanten mit a_1, \dots, a_m markiert sind.



Da $|Q| = n$, können die Zustände p_0, p_1, \dots, p_n nicht alle verschieden sein (da $n + 1$ Zustände), also gibt es $k, l \in \{0, 1, \dots, n\}$ mit $k < l$, sodass $p_k = p_l$.

Wir setzen $u := a_1 \dots a_k$, $v := a_{k+1} \dots a_l$, $w := a_{l+1} \dots a_m$.

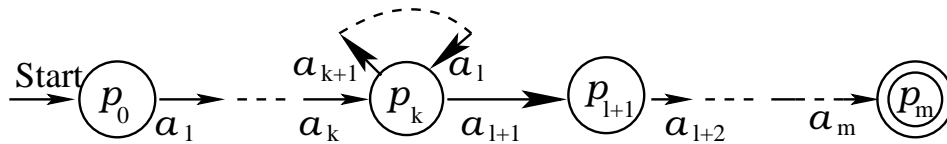
Offenbar ist $x = uvw$, $|uv| = l \leq n$, $|v| = l - k \geq 1$. Es bleibt nur noch die „Pump-Eigenschaft“ nachzukontrollieren.

Wie zerschneiden den Weg in drei Teile:

P_u von q_0 nach p_k , Kantenmarkierung a_1, \dots, a_k ;

P_v von p_k nach $p_l = p_k$, Kantenmarkierung a_{k+1}, \dots, a_l ;

P_w von p_l nach p_m , Kantenmarkierung a_{l+1}, \dots, a_m ;



Sind P, P' zwei Wege, deren Endpunkt p und Anfangspunkt p' übereinstimmen, so bezeichnet PP' die Verkettung der beiden.

\Rightarrow

$P_u P_w$ ist Weg von q_0 nach p_m mit Kantenmarkierung uw ;

$P_u P_v P_v P_w$ ist Weg von q_0 nach p_m mit Kantenmarkierung uv^2w ;

allgemein:

für $i \geq 0$ ist $P_u \underbrace{P_v \dots P_v}_{i\text{-mal}} P_w$ Weg von q_0 nach p_m mit Kantenmarkierung $uv^i w$. Weil $p_m \in F$, erhalten wir $uv^i w \in L$ für $i \geq 0$.

Das Pumping Lemma kann zum Nachweis der Nichtregularität von Mengen genutzt werden. Dazu kann folgendes Schema genutzt werden:

1. (Wörtlich) Beweis indirekt. Annahme: L ist regulär.
2. (Wörtlich) Dann gibt es ein $n \geq 1$ mit den im Pumping-Lemma behaupteten Eigenschaften.
3. (Problemspezifisch) Wähle $x \in L$ mit $|x|$ genügend groß.
4. (Wörtlich) Gemäß Pumping-Lemma kann man $x = uvw$ mit $|uv| \leq n$ und $|v| \geq 1$ schreiben, sodass $X = \{uw, uvw, uv^2w, \dots\} \subseteq L$ (problemspezifisch) mit $u = \dots, v = \dots$ und $w = \dots$.
5. (Problemspezifisch) Wähle ein passendes Element y aus X aus und zeige direkt, dass y nicht in L sein kann.
(Manchmal ist das uw und manchmal ist das $uv^i w$ für ein $i \geq 2$.)
Dies ist der gewünschte Widerspruch.

Beispiel 4.7: Mit Hilfe des Pumping-Lemmas beweise man, dass

$L = \{a^{2m} b^m \mid m \in \mathbb{N}\}$ keine reguläre Sprache ist.

1. Beweis indirekt. Annahme: L ist regulär.
2. Dann gibt es ein $n \geq 1$ mit den im Pumping-Lemma behaupteten Eigenschaften.
3. Wähle $x \in L$ mit $|x|$ genügend groß: $x = a^{2n} b^n$ mit $n \geq 1$, beliebig $\Rightarrow |x| = 3n$.
4. Gemäß Pumping-Lemma kann man $x = uvw$ mit $|uv| \leq n$ und $|v| \geq 1$ schreiben als $x = a^{|u|} a^{|v|} a^{n-|u|-|v|} a^n b^n$.
Die Zerlegung erfolgt nur in a^n , da $|uv| \leq n$ gelten muss.
5. Für $i = 0$ folgt $a^{n-|v|} a^n b^n = a^{2n-|v|} b^n \notin L$, da $|v| \neq 0$.
Dies ist der gewünschte Widerspruch.

Beispiel 4.8: Man beweise, dass die Sprache L

$L = \{a^{n^2} \mid n \geq 1\} = \{a, a a a a, \dots\}$ nicht regulär ist.

Angenommen, L sei Sprache eines FA mit $|Q| = n_0$.

Dann müsste sich jedes Wort $x \in L$ mit einer Länge $\geq n_0$ in $x = uvw$ mit $v \neq \varepsilon$ und $|uv| \leq n_0$ zerlegen lassen.

x sei $a^{n_0^2}$ mit der Zerlegung $a^{n_0^2} = uvw$ mit $1 \leq |v| \leq n_0$ und $|uv| \leq n_0$ mit $u v^i w \in L$ für alle $i \in \mathbb{N}$. Für $i = 2$ folgt $u v v w \in L$.

Es gilt aber:

1. $u v v w$ hat mindestens die Länge $n_0^2 + 1$, da $|u v v w| = |u v w| + |v| \geq n_0^2 + 1$.
2. $u v v w$ hat höchstens die Länge $n_0^2 + n_0$ ($|v| \leq n_0$), wobei $n_0^2 + n_0 < (n_0 + 1)^2 = n_0^2 + 2 \cdot n_0 + 1$.
 $\Rightarrow n_0^2 < |u v v w| < (n_0 + 1)^2$,

d.h. die Länge des Wortes $u v v w$ ist kein Quadrat einer natürlichen Zahl,

d.h. $u v v w$ kann nicht zur Sprache gehören,

d.h. L ist nicht Sprache eines FA wie angenommen und damit nicht regulär.

4.6 Abschlusseigenschaften für reguläre Sprachen

d.h., es werden Operationen angegeben, die aus regulären Sprachen wieder reguläre Sprachen erzeugen.

Satz 4.7: Alle endlichen Sprachen sind regulär

Satz 4.8: Die regulären Sprachen sind abgeschlossen unter Vereinigung, Verkettung (Konkatenation), KLEENEscher Hüllenbildung, Komplement- und Schnittbildung.

Sei $\Sigma = \{a_1, \dots, a_n\}$ und sei Δ ein Alphabet.

Eine **Substitution** ist eine Funktion $f : \Sigma \rightarrow \mathcal{P}(\Delta^*)$, d.h. $f(a_i)$ ist eine Sprache über Δ , für $1 \leq i \leq n$.

Für $w = b_1 \dots b_m \in \Sigma^*$ setzen wir $f(w) := \underbrace{f(b_1)f(b_2)\dots f(b_m)}_{\text{Konkatenation von Sprachen}}$ und für

$L \subseteq \Sigma^*$ setzen wir $f(L) := \bigcup \{f(w) \mid w \in L\}$.

Siehe **Beispiel 5.2:** Menge aller Bezeichner $L_1 = L(B(B + Z)^*)$ über dem Alphabet $\Sigma = \{B, Z\}$ mit $f(B) = \{a, b, c, \dots, z\}$ und $f(Z) = \{0, 1, \dots, 9\}$.

Satz 4.9: Sei f Substitution wie oben beschrieben. Sind $f(a_1), \dots, f(a_n)$ reguläre Sprachen und ist L regulär, so ist auch $f(L)$ regulär.

f heißt **Homomorphismus**, wenn f eine Substitution ist mit $|f| = 1$ für alle $a \in \Sigma$. Ist L eine Sprache, f Homomorphismus, so ist

$$f(L) = \{f(b_1), \dots, f(b_m) \mid b_1 \dots b_m \in L\}.$$

Ist weiter $L' \subseteq \Delta^*$, so kann man das Urbild $f^{-1}(L') = \{w \in \Sigma^* \mid f(w) \in L'\}$ betrachten („inverser Homomorphismus“).

Satz 4.10: \mathcal{L}_{reg} ist abgeschlossen unter Homomorphismen und inversen Homomorphismen:

Sei $f : \Sigma \rightarrow \Delta^*$ ein Homomorphismus. Dann gilt:

- (a) Ist $L \subseteq \Sigma^*$ regulär, so ist auch $f(L)$ regulär.
- (b) Ist $L' \subseteq \Delta^*$ regulär, so ist auch $f^{-1}(L')$ regulär.

4.7 Entscheidbarkeitsfragen für reguläre Sprachen

Die Frage ist, ob eine reguläre Sprache eine bestimmte Eigenschaft hat oder nicht.

Folgende Probleme für reguläre Sprachen sind entscheidbar:

- (a) Ist $L = \emptyset$?
- (b) Ist $|L| < \infty$?
- (c) Ist $L = \Sigma^*$?
- (d) Ist $L_1 \subseteq L_2$?
- (e) Ist $L_1 = L_2$?
- (f) Ist $\varepsilon \in L$?
- (g) Ist $w \in L$? (für gegebenes $w \in \Sigma^*$)

Satz 4.11: Die Menge von Wörtern (Sprache), die von einem endlichen Automaten M mit n Zuständen akzeptiert wird, ist genau dann

- **nichtleer**, wenn M ein Wort der Länge kleiner als n akzeptiert;
- **unendlich**, wenn M ein Wort der Länge l akzeptiert mit $n \leq l < 2n$.

Kapitel 5

Grammatiken und Chomsky-Hierarchie

5.1 Grammatik

Reguläre Ausdrücke stellen eine Möglichkeit der generativen Definition von Sprachen dar, die jedoch nicht für alle Sprachen anwendbar ist.

Das allgemeinste (und üblichste) Werkzeug zur (generativen) Definition von Sprachen ist die Grammatik.

Beispiel 5.1: Umgangssprache

$\langle \text{Satz} \rangle^1 ::= \langle \text{Subjekt} \rangle \langle \text{Prädikat} \rangle \langle \text{Objekt} \rangle$

$\langle \text{Subjekt} \rangle ::= \langle \text{Artikel} \rangle \langle \text{Attribut} \rangle \langle \text{Substantiv} \rangle$

$\langle \text{Objekt} \rangle ::= \langle \text{Artikel} \rangle \langle \text{Attribut} \rangle \langle \text{Substantiv} \rangle$

$\langle \text{Artikel} \rangle ::= \text{der} \mid \text{die} \mid \text{das} \mid \varepsilon$

$\langle \text{Attribut} \rangle ::= \varepsilon \mid \langle \text{Adjektiv} \rangle \mid \langle \text{Adjektiv} \rangle \langle \text{Attribut} \rangle$

$\langle \text{Adjektiv} \rangle ::= \text{kleine} \mid \text{bissige} \mid \text{große}$

$\langle \text{Substantiv} \rangle ::= \text{Hund} \mid \text{Katze}$

$\langle \text{Prädikat} \rangle ::= \text{jagt}$

1) Platzhalter für syntaktische Einheiten: $\langle \dots \rangle$

Satz \Rightarrow der kleine bissige Hund jagt die große Katze

Diese Grammatik liefert eine unendliche Sprache:

z.B.: Satz \Rightarrow der kleine Hund jagt die große große ... große Katze

Bisher wurden Sprachen in Verbindung mit Endlichen Automaten betrachtet, wobei aber noch keine Aussage darüber gemacht werden kann, wie Sprachen bezüglich welcher Kriterien systematisiert werden können und welche Sprache von welchem Automatentyp akzeptiert werden kann.

Wenn man die Produktionen der erzeugenden Grammatik einer Sprache betrachtet, dann ist es möglich, eine derartige Systematik zu erhalten.

Def. 5.1: Eine allgemeine **Grammatik** ist ein Tupel $G = (V, \Sigma, P, S)$ mit:

Σ **Alphabet**;

V **Vokabular**, eine endliche nichtleere Menge von Zeichen mit $\Sigma \cap V = \emptyset$;

P endliche **Produktionenmenge** mit $P \subset ((V \cup \Sigma)^+ - \Sigma^*) \times (V \cup \Sigma)^*$;

S **Satzsymbol**, $S \in V$.

Die Elemente von Σ werden Terminalzeichen genannt, die Elemente von V Variable.

Eine Grammatik ist eine endliche Erzeugungsvorschrift für eine (i.A. unendliche) Sprache.

Die Produktionen sind als Ersetzungsregeln anzusehen.

Dabei kann in einer Zeichenfolge eine zusammenhängende Teilfolge, die der linken Seite einer Produktion entspricht, durch die rechte Seite derselben Produktion ersetzt werden.

Wir wollen eine Produktion (l, r) auch in der Form $l \rightarrow r$ oder $l ::= r$ schreiben.

Def. 5.2: Es sei $G = (V, \Sigma, P, S)$ eine Grammatik und $w, w' \in (\Sigma \cup V)^*$.

1. Dann erzeugt w in G direkt w' , in Zeichen: $w \xrightarrow{G} w'$, wenn $w = xly$ und $w' = xry$ mit $x, y \in (\Sigma \cup V)^*$ und $(l, r) \in P$.

In Worten: w' ist in 1 Schritt aus w ableitbar.

2. \xrightarrow{G}^* ist die reflexive transitive Hülle von \xrightarrow{G} D.h. $\alpha \xrightarrow{G}^* \alpha'$,

3. Die von der Grammatik $G = (V, \Sigma, P, S)$ erzeugte Sprache $L(G)$ ist die Menge $L(G) = \{w \in \Sigma^* \mid S \xrightarrow{G}^* w\}$, wobei \xrightarrow{G}^* die reflexive transitive Hülle von \xrightarrow{G} ist.

Grammatik 5.1:

$V = \{S, A, B, C\}$, (S ist das Satzsymbol)

$\Sigma = \{a, b\}$,

P: $(S, bC)_1, (S, aA)_2, (C, bS)_3, (C, aB)_4, (A, aS)_5, (A, bB)_6, (B, bA)_7,$
 $(B, aC)_8, (S, \varepsilon)_9$

Mögliche Ableitungen:

$S \xrightarrow[G]{1} bC \xrightarrow[G]{4} baB \xrightarrow[G]{8} baaC \xrightarrow[G]{3} baabS \xrightarrow[G]{9} baab.$

$S \xrightarrow[G]{1} bC \xrightarrow[G]{3} bbS \xrightarrow[G]{1} bbbC \xrightarrow[G]{3} bbbbS \xrightarrow[G]{9} bbbb.$

usw.

$\Rightarrow: L(G) = \{x \mid x \in \{a, b\}^* \text{ mit } |x|_a \text{ und } |x|_b \text{ ist gerade}\}$

Grammatik 5.2:

$V = \{S, B, C\}$, mit S Satzsymbol

$\Sigma = \{a, b, c\}$,

P: $(S, aSBC)_1, (S, aBC)_2, (CB, BC)_3, (aB, ab)_4, (bB, bb)_5, (bC, bc)_6,$
 $(cC, cc)_7$

Eine Ableitung:

$S \xrightarrow[G]{1} aSBC \xrightarrow[G]{1} aaSBCBC \xrightarrow[G]{2} aaaSBCBCBC \xrightarrow[G]{3} aaaSBBCCBC \xrightarrow[G]{3} aaaSBBBCCC \xrightarrow[G]{4} aaaSBBBCCC \xrightarrow[G]{5} aaaSBBBCCC \xrightarrow[G]{6} aaaSBBBCCC \xrightarrow[G]{7} aaaSBBBCCC = a^3b^3c^3.$

Vermutung: $L = \{a^n b^n c^n \mid n \geq 1\}$ gilt. Dies lässt sich auch beweisen.

5.2 Chomsky-Hierarchie

Es können nun nach Chomsky verschiedene Typen von Grammatiken unterschieden werden:

Def. 5.3: Eine Grammatik gemäß allgemeiner Definition ist vom **Chomsky-Typ 0** und heißt allgemeine Grammatik.

Def. 5.4: Eine Grammatik $G = (V, \Sigma, P, S)$ ist genau dann vom **Chomsky-Typ 1** (kontextsensitive Grammatik), wenn das Produktionensystem genau eine der folgenden Bedingungen erfüllt:

1.) $\forall p$ mit $p = (l, r) \in P$ gilt:

$l = x_1 z x_2, r = x_1 w x_2$, mit $x_1, x_2 \in (V \cup \Sigma)^*$, $z \in V, w \in (V \cup \Sigma)^+$.
 p ist längenmonoton, d. h. $|l| \leq |r|$.

2.) Ausnahme: $S \rightarrow \varepsilon$. Gibt es diese Produktion, dann darf das Satzsymbol in keiner weiteren Produktion auf der rechten Seite vorkommen und für jede davon verschiedene Produktion gilt 1.).

Def. 5.5: Eine Grammatik $G = (V, \Sigma, P, S)$ heißt **(wortlängen-)monoton**, wenn das Produktionensystem genau eine der folgenden Bedingungen erfüllt:

1.) $\forall p$ mit $p = (l, r) \in P$ gilt: $|l| \leq |r|$.

2.) Ausnahme: $S \rightarrow \varepsilon$. Gibt es diese Produktion, dann darf das Satzsymbol in keiner weiteren Produktion auf der rechten Seite vorkommen und für jede davon verschiedene Produktion gilt 1.).

Bemerkung: Aus der Forderung $w \in (V \cup \Sigma)^+$ in der rechten Seite der kontextsensitiven Grammatiken folgt unmittelbar die Längenmonotonie.

Umgekehrt kann jede längenmonotone Produktion leicht durch Einführung von neuen Variablen in eine Menge von kontextabhängigen Produktionen überführt werden. Es gilt, dass monotone Grammatiken nur kontextsensitive Sprachen beschreiben können. I. Allg. sind diese Grammatiken vom Typ 0.

Def. 5.6: Eine Grammatik $G = (V, \Sigma, P, S)$ ist genau dann vom **Chomsky-Typ 2** und heißt **kontextfrei**, wenn $P \subseteq V \times (V \cup \Sigma)^*$ ist, d. h. jede Produktion ist von der Form (z, w) mit $z \in V, w \in (V \cup \Sigma)^*$.

Def. 5.7: Eine Grammatik $G = (V, \Sigma, P, S)$ ist genau dann vom **Chomsky-Typ 3** und heißt **regulär**, wenn entweder

$P \subseteq V \times (\Sigma V \cup \{\varepsilon\})$ oder $P \subseteq V \times (V \Sigma \cup \{\varepsilon\})$

ist, d.h. jede Produktion ist von der Form

- entweder $[(z, a z') \text{ oder } (z, \varepsilon) \text{ (rechtslinear)}]$ mit $z, z' \in V, a \in \Sigma$

- oder $[(z, z' a) \text{ oder } (z, \varepsilon) \text{ (linkslinear)}]$ mit $z, z' \in V, a \in \Sigma$.

Grammatik 5.3: Gegeben ist folgende Grammatik G :

$V = \{S, A, B, C\}$, mit S Satzsymbol und $\Sigma = \{a, b\}$,

$P : (S \rightarrow bC)_1, (S \rightarrow aA)_2, (C \rightarrow bS)_3, (C \rightarrow aB)_4, (A \rightarrow aS)_5,$
 $(A \rightarrow bB)_6, (B \rightarrow bA)_7, (B \rightarrow aC)_8, (S \rightarrow \varepsilon)_9,$

\Rightarrow

δ	Eingabe	
	a	b
q_0	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

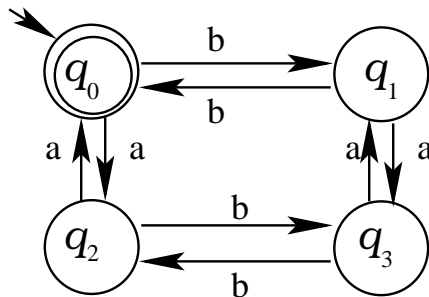


Bild 5.1: Zustandsüberföhrungsfunktion δ und Automatengraph $M(G)$

$L(G) = \{w \in \{a, b\}^* \mid |w|_a \text{ ist gerade und } |w|_b \text{ ist gerade} \}$

Def. 5.8: Eine Sprache L heit vom Chomsky-Typ i ($i = 0, 1, 2, 3$), falls es eine Grammatik G vom Chomsky-Typ i gibt, so dass $L = L(G)$.

Sei nun \mathcal{L}_i die Klasse der Sprachen vom CHOMSKY-Typ i , so gelten uneingeschrnkt folgende Inklusionen:

$$\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0.$$

Dabei stellt die allgemeinste Klasse \mathcal{L}_0 der mittels Grammatiken beschreibbaren Sprachen (= Menge der rekursiv aufzhlbaren Sprachen) nur eine echte Teilmenge aller Sprachen dar.

Die Menge der entscheidbaren (rekursiven) Sprachen ist eine echte Teilmenge der Klasse \mathcal{L}_0 , aber zugleich eine echte Obermenge von \mathcal{L}_1 .

Kapitel 6

Kontextfreie Grammatiken und kontextfreie Sprachen

6.1 BACKUS-NAUR-Form

Eine andere Darstellung für kontextfreie Grammatiken ist die sogenannte **BACKUS-NAUR-Form**:

Jede Produktion (z, v) mit $z \in V$, $v \in (V \cup \Sigma)^*$ wird dargestellt als $z ::= v$.

Enthält die Menge $P_z = \{(z_i, v_i) \in P \mid z_i = z, i \geq 1\}$ für ein $z \in V$ mehr als eine Produktion, ist es üblich, die gesamte Menge P_z mit $|P_z| = n_z$ in einer Zeile („Metaregel“) wie folgt darzustellen: $z ::= v_1 \mid v_2 \mid \dots \mid v_{n_z}$.

Üblicherweise werden auch Variable zur besseren Unterscheidung in spitze Klammern eingeschlossen: $\langle z \rangle$.

In erweiterter BNF (EBNF) sind noch folgende Notationen erlaubt:

$z ::= u [v] w$ für ein Produktionenpaar $(z, u w), (z, u v w)$.

Das Wort v kann zwischen u und w auftreten, muss aber nicht;

$z ::= u \{v\} w$ für Produktionen $(z, u w), (z, u z' w), (z', v), (z', v z')$.

Das Wort v kann beliebig oft wiederholt werden oder gar nicht auftreten (entspricht dem KLEENE-Stern in regulären Ausdrücken).

Das Satzsymbol ist die Variable, die in der ersten Produktion auf der linken Seite steht. Z.B.:

$S ::= b C \mid a A \mid \varepsilon$

$A ::= a S \mid b B$

$B ::= b A \mid a C$

$C ::= b S \mid a B$

eigentlich rechtslinear
aber rl.G. \subset kf. G.

6.2 Chomsky-Normalform

Def. 6.1: Eine kontextfreie Grammatik $G = (V, \Sigma, S, P)$ ist in Chomsky-Normalform, falls

1. entweder alle Produktionen von G haben die Form

$$A \rightarrow BC \quad (A, B, C \in V) \text{ oder } A \rightarrow a \quad (A \in V, a \in \Sigma)$$

2. oder alle Produktionen von G haben die Form

$$A \rightarrow BC \text{ oder } A \rightarrow a \text{ oder } S \rightarrow \varepsilon$$

und S darf in keiner Produktion auf der rechten Seite vorkommen.

6.3 Beispiele und Ableitungen

Grammatik 6.1:

$V = \{S\}$, mit S Satzsymbol und $\Sigma = \{a, b\}$,

$P : (S \rightarrow \varepsilon)_1, (S \rightarrow a S b)_2$

Mögliche Ableitungen:

1. $S \xrightarrow[G]{1} \varepsilon$

2. $S \xrightarrow[G]{2} a S b \xrightarrow[G]{2} a a S b b \xrightarrow[G]{1} a a b b$

3. $S \xrightarrow[G]{2} a S b \xrightarrow[G]{2} a a S b b \xrightarrow[G]{2} \dots \xrightarrow[G]{2} \underbrace{a \dots a}_{n \text{ mal}} S \underbrace{b \dots b}_{n \text{ mal}} \xrightarrow[G]{1} a^n b^n$

Grammatik 6.2:

$V = \{S\}$, mit S Satzsymbol und $\Sigma = \{0, 1\}$,

$P : (S \rightarrow \varepsilon)_1, (S \rightarrow S S)_2, (S \rightarrow 0 S 1)_3$

Eine mögliche Ableitung:

$$\begin{aligned} S &\xrightarrow[G]{2} S S \xrightarrow[G]{2} S S S \xrightarrow[G]{3} 0 S 1 S S \xrightarrow[G]{1} 0 1 S S \xrightarrow[G]{3} 0 1 S 0 S 1 \xrightarrow[G]{2} 0 1 S 0 S S 1 \\ &\xrightarrow[G]{1} 0 1 0 S S 1 \xrightarrow[G]{3} 0 1 0 0 S 1 S 1 \xrightarrow[G]{1} 0 1 0 0 1 S 1 \xrightarrow[G]{1} 0 1 0 0 1 0 1 1 \end{aligned}$$

Für $0 = „(“$ und $1 = „)“$ ist $L(G)$ die Menge der korrekten Klammerausdrücke.

$\alpha \xrightarrow[G]{*} \alpha'$ hieß nach Def.5.2: α' ist in beliebig vielen Schritten aus α ableitbar.

Die Wörter $\alpha \in (V \cup \Sigma)^*$, die man so aus S erzeugen kann, heißen **Satzformen** in G .

6.4 Ableitungsbäume, Linksableitung, Rechtsableitung

Ableitungsbäume, auch Syntaxbäume genannt, sind ein Hilfsmittel in der Compilertechnik für die Analyse von Programmtexten.

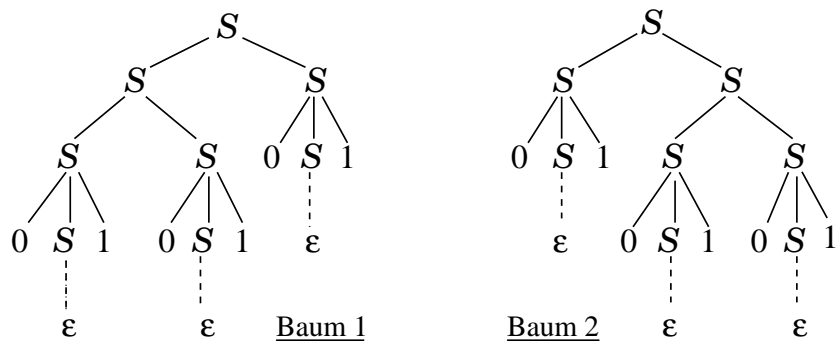
Ableitungen werden als Bäume dargestellt.

Beispiel Grammatik 6.2

Eine mögliche Ableitung:

$$\begin{aligned}
 S &\xrightarrow[G]{2} S S \xrightarrow[G]{2} S S S \xrightarrow[G]{3} 0 S 1 S S \xrightarrow[G]{1} 0 1 S S \xrightarrow[G]{3} 0 1 S 0 S 1 \xrightarrow[G]{1} 0 1 S 0 1 \\
 &\xrightarrow[G]{3} 0 1 0 S 1 0 1 \xrightarrow[G]{1} 0 1 0 1 0 1,
 \end{aligned}$$

die folgende Ableitungsbäume liefert:



Eine andere mögliche Ableitung:

$$\begin{aligned}
 S &\xrightarrow[G]{2} S S \xrightarrow[G]{3} S 0 S 1 \xrightarrow[G]{2} S S 0 S 1 \xrightarrow[G]{3} 0 S 1 S 0 S 1 \xrightarrow[G]{1} 0 S 1 S 0 1 \\
 &\xrightarrow[G]{3} 0 S 1 0 S 1 0 1 \xrightarrow[G]{1} 0 S 1 0 1 0 1 \xrightarrow[G]{1} 0 1 0 1 0 1,
 \end{aligned}$$

die ebenfalls mehrere Ableitungsbäume liefert.

Def. 6.2: Sei $G = (V, \Sigma, S, P)$ eine kontextfreie Grammatik.

a) Ein **Ableitungsbaum** T , auch Syntaxbaum genannt, ist ein gerichteter, geordneter¹ Baum mit Wurzel, dessen Knoten mit je einem Buchstaben aus $V \cup \Sigma$ oder ε beschriftet sind, wobei folgendes gilt:

(I) Die Wurzel ist mit S beschriftet,

(II) Ist v ein Knoten, der mit $a \in \Sigma$ oder mit ε beschriftet ist, so ist v ein Blatt, hat also keinen Nachfolgerknoten.

(III) Ist v ein Knoten, der mit $A \in V$, beschriftet ist, und ist v kein Blatt, so gilt:

(i) die Nachfolgerknoten v_1, \dots, v_r von v sind mit $X_1, \dots, X_r \in \Sigma \cup V$ beschriftet und $A \rightarrow X_1 \dots X_r$ ist Produktion in P

¹Ein Baum heißt geordnet, wenn die unmittelbaren Nachfolgerknoten eines Knotens eine Reihenfolge „von links nach rechts“ haben

oder

- (ii) v hat genau einen Nachfolgerknoten v' , der mit ε beschriftet ist und $A \rightarrow \varepsilon$ ist Produktion in P .

- b) Ist T ein Ableitungsbaum, so bezeichnen wir mit $\alpha(T)$ das Wort über $\Sigma \cup V$, das sich beim Lesen der Blätter von T von links nach rechts ergibt. ($\alpha(T)$ heißt Ergebnis oder Resultat oder Blattwort von T .)

Bei Ableitungsfolgen ist die Eindeutigkeit der Anwendung der Produktionen nicht immer gegeben (siehe Bsp.). Diese Information kann in Ableitungsbäumen enthalten sein. Um eindimensionale (zeigerfreie) Notationen zu erhalten, erzeugt man **Linksableitungen** (bzw. Rechtsableitungen).

In einer Linksableitung gilt für jeden Ableitungsschritt, dass immer auf die am weitesten links stehende Variable eine Produktion angewendet wird (Analog Rechtsableitung).

Für **Grammatik 6.2** sieht eine Linksableitung dann so aus:

$$S \xrightarrow[G]{2} S S \xrightarrow[G]{2} S S S \xrightarrow[G]{3} 0 S 1 S S \xrightarrow[G]{1} 0 1 S S \xrightarrow[G]{3} 0 1 0 S 1 S \xrightarrow[G]{1} 0 1 0 1 S \xrightarrow[G]{3} 0 1 0 1 0 S 1 \xrightarrow[G]{1} 0 1 0 1 0 1, \text{ dazu gehört Baum 1.}$$

Zu Baum 2 gehört eine andere Linksableitung:

$$S \xrightarrow[G]{2} S S \xrightarrow[G]{3} 0 S 1 S \xrightarrow[G]{1} 0 1 S \xrightarrow[G]{2} 0 1 S S \xrightarrow[G]{3} 0 1 0 S 1 S \xrightarrow[G]{1} 0 1 0 1 S \xrightarrow[G]{3} 0 1 0 1 0 S 1 \xrightarrow[G]{1} 0 1 0 1 0 1.$$

Eine kontextfreie Grammatik, für die es ein Wort $w \in L(G)$ gibt, das (mindestens) zwei verschiedene Ableitungsbäume besitzt, heißt **mehrdeutig**. (Siehe unser Beispiel.)

Sie heißt **eindeutig**, wenn jedes Wort $w \in L(G)$ genau einen Ableitungsbaum besitzt.

Eine Sprache $L \in \mathcal{L}$ heißt **inhärent mehrdeutig**, wenn jede kontextfreie Grammatik G mit $L = L(G)$ mehrdeutig ist.

Satz 6.1: Die Klasse \mathcal{L}_2 (CFL) der kontextfreien Sprachen ist abgeschlossen unter Vereinigung, Konkatenation und Kleene-Stern (reflexive transitive Hülle). Sie ist nicht abgeschlossen unter Schnitt und Komplement.

Beweis erfolgt durch Konstruktion:

Seien dazu $L_1, L_2 \in \mathcal{L}_2$ mit $L_1 = L(G_1)$, $L_2 = L(G_2)$, wobei $G_1 = (V_1, \Sigma, P_1, S_1)$ und $G_2 = (V_2, \Sigma, P_2, S_2)$ kontextfreie Grammatiken mit $V_1 \cap V_2 = \emptyset$ sind.

Ferner sei S ein neues nichtterminales Symbol ($S \notin V_1 \cup V_2$).

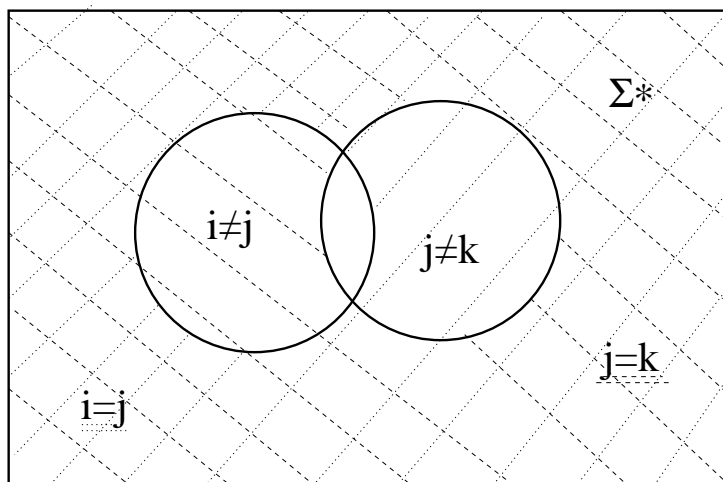
Vereinigung: $L_1 \cup L_2 = L(G)$ mit $G = (V, \Sigma, P, S)$ mit
 $V = V_1 \cup V_2 \cup \{S\}$,
 $P = P_1 \cup P_2 \cup \{S ::= S_1 \mid S_2\}$.

Konkatenation: $L_1 \otimes L_2 = L(G)$ mit $G = (V, \Sigma, P, S)$ mit
 $V = V_1 \cup V_2 \cup \{S\}$,
 $P = P_1 \cup P_2 \cup \{S ::= S_1 S_2\}$.

KLEENE-Stern: $L_1^* = L(G)$ mit $G = (V, \Sigma, P, S)$ mit
 $V = V_1 \cup \{S\}$,
 $P = (P_1 - \{S_1 ::= \varepsilon\}) \cup \{S ::= \varepsilon \mid S_1, S_1 ::= S_1 S_1\}$.

Schnitt: z.B. $L_1 = \{a^i b^i c^j \mid i, j > 0\}$, $L_2 = \{a^i b^j c^j \mid i, j > 0\}$;
 $L_1 \cap L_2 = \{a^i b^i c^i \mid i > 0\} \notin \mathcal{L}_2$.

Komplement: z.B. $L_1 = \{a^i b^j c^k \mid i \neq j \vee j \neq k, i, j, k > 0\}$;
 $\overline{L_1} = \{a^i b^i c^i \mid i > 0\} \notin \mathcal{L}_2$



Kapitel 7

Kellerautomaten

7.1 Allgemeiner Kellerautomat, PDA (englisch: pushdown automaton).

Wir erweitern das Konzept des endlichen Automaten um einen weiteren Speicher, sodass wir einen Automatentyp erhalten, der genau zur Erkennung kontextfreier Sprachen geeignet ist.

Def. 7.1: Ein Kellerautomat ist ein Tupel $K = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$ mit

Q endliche Menge von **Zuständen**,

Σ **(Eingabe-)Alphabet**,

Γ **Kelleralphabet**,

$\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \mathcal{P}_{<\infty}(Q \times \Gamma^*)$ **Zustandsüberföhrungsfunktion**
(dabei bedeutet $\mathcal{P}_{<\infty}(M)$ die Menge aller endlichen Teilmengen von M),

$q_0 \in Q$ **Anfangs- oder Start- oder Initialzustand**,

\perp **Kellerendezeichen oder Kelleranfangszeichen**,

$F \subseteq Q$ **Menge von Endzuständen**, (F ist oft nicht ausgezeichnet).

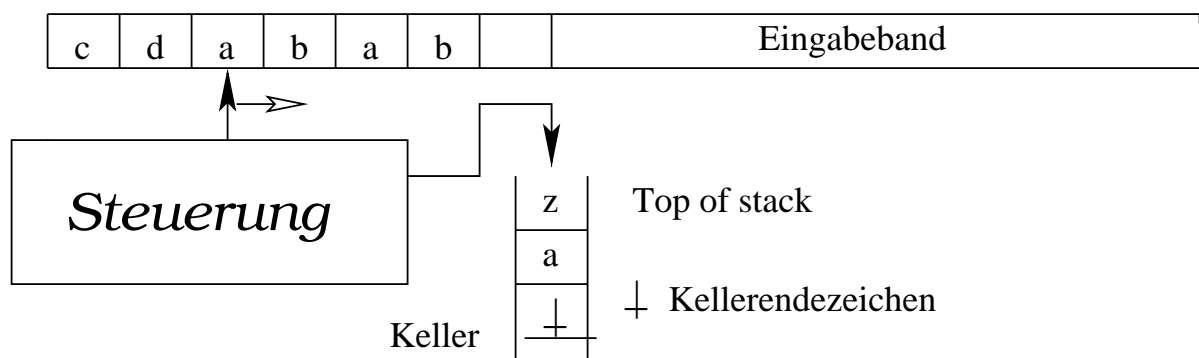


Bild 7.1: allgemeiner Kellerautomat PDA

Wie erfolgt die Überführung?

$(q_j, \gamma_w) \in \delta(q_i, x, \gamma)$ mit $q_i, q_j \in Q$, $\gamma \in \Gamma$, $\gamma_w \in \Gamma^*$, $x \in \Sigma \cup \{\varepsilon\}$

Eingabeband:

- für $x \in \Sigma$: Zeichen unter dem Lesekopf wird gelesen,
Rechtsbewegung des Lesekopfes
- für $x = \varepsilon$: Zeichen unter dem Lesekopf wird ignoriert,
Lesekopf bleibt stehen

PDA K geht von q_i nach q_j über.

Keller:

Im Keller wird das oberste Kellerzeichen γ gelöscht und das Kellerwort γ_w wird eingetragen:

- $\gamma_w = \varepsilon$: da das oberste Zeichen gelöscht wurde, liegt das nächste Kellerzeichen oben,
- $\text{lgth}(\gamma_w) = 1$: das Zeichen γ_w wird eingetragen,
- $\text{lgth}(\gamma_w) > 1$: das Wort γ_w wird so in den Keller eingetragen, dass das erste abzuarbeitende Zeichen oben liegt.

!!! Für $\delta(q_i, x, \gamma) = \emptyset$ oder $\delta(q_i, \varepsilon, \gamma) = \emptyset$ hält der PDA K an !!!

(\emptyset : Folgezustand nicht definiert)

Bemerkung: Die so definierten PDA's sind per Definition nichtdeterministisch \Rightarrow NPDA (engl. nondeterministic pushdown automaton).

Die Beschreibung eines PDA K zu einem bestimmten Zeitpunkt erfolgt über die Konfiguration von K .

Eine Konfiguration des PDA K ist ein Tripel $k = (q, x, \gamma_w)$ mit

$k = (q, x, \gamma_w) \in Q \times \Sigma^* \times \Gamma^*$, wobei $x \in \Sigma^*$ der noch zu lesende Rest des Eingabebandes ist und $\gamma_w \in \Gamma^*$ der Kellerinhalt.

Auf der Menge der Konfigurationen definiert die Überföhrungsfunktion δ

- eine Nachfolgerrelation ($k \vdash_K t$):
 $k = (q_i, x w, \gamma \gamma_w)$ führt unmittelbar zu $t = (q_j, w, \gamma_w' \gamma_w)$
falls $(q_j, \gamma_w') \in \delta(q_i, x, \gamma)$ und
- ($k \vdash_K^* t$) als reflexive transitive Hölle von ($k \vdash_K t$): k führt zu t , falls $k = t$ oder es existiert eine Folge von Konfigurationen $k = k_0, k_1, \dots, k_j = t$ mit $k_{i-1} \vdash_K k_i$ ($i = 1, \dots, j$).

Def. 7.2: Die von einem PDA K **unter Leeren des Kellers akzeptierte Sprache** $L(K)$ ist definiert durch

$$L(K) = \{x \in \Sigma^* \mid (q_0, x, \perp) \vdash_K^* (q, \varepsilon, \varepsilon) \text{ mit } q \in Q\}.$$

Diese Definition ist die im Allgemeinen übliche, da in der Compilertechnik diese Kellerautomaten genutzt werden.

Theoretisch gibt es zwei weitere Definitionen:

Def. 7.3: Die von einem PDA K **durch Erreichen des Endzustandes akzeptierte Sprache** $L(K)$ ist definiert durch

$$L(K) = \{x \in \Sigma^* \mid (q_0, x, \perp) \vdash_K^* (q, \varepsilon, \gamma) \text{ mit } q \in F, \gamma \in \Gamma^*\}.$$

Def. 7.4: Die von einem PDA K **durch Erreichen des Endzustandes unter Leeren des Kellers akzeptierte Sprache** $L(K)$ ist definiert durch

$$L(K) = \{x \in \Sigma^* \mid (q_0, x, \perp) \vdash_K^* (q, \varepsilon, \varepsilon) \text{ mit } q \in F\}.$$

Man kann zeigen, dass für den **nichtdetermin.** PDA gilt:

$$\begin{aligned} \{L(K(\text{Leeren des Kellers}))\} &= \{L(K(\text{Erreichen des Endzustandes}))\} \\ &= \{L(K(\text{Leeren des Kellers und Erreichen des Endzustandes}))\}. \end{aligned}$$

Satz 7.1: Wird eine Sprache L von einem NPDA K durch leeren Keller erkannt ($L = L(K)$), so ist L kontextfrei.

Satz 7.2: Für jede kontextfreie Sprache L gibt es einen NPDA K mit $L = L(K)$.

Folgerung: Die Klasse der von NPDA akzeptierbaren Sprachen ist identisch mit der Klasse \mathcal{L}_2 (Klasse der kontextfreien Sprachen).

7.2 Deterministischer Kellerautomat, DPDA

Ein PDA $K = (\Sigma, \Gamma, Q, \delta, \perp, q_0, F)$ heißt deterministisch, wenn für jedes Paar $\delta(q, x, \gamma)$ mit $q \in Q$, $x \in \Sigma$, $\gamma \in \Gamma$ gilt:

$$|\delta(q, x, \gamma)| + |\delta(q, \varepsilon, \gamma)| \leq 1,$$

dh. die Überführung ist entweder für eine Eingabe $x \in \Sigma$ oder für ε bei demselben Zustand $q \in Q$ und demselben Kellerbuchstaben $\gamma \in \Gamma$ oder weder für $x \in \Sigma$ noch für ε definiert.

DPDA's werden über die Akzeptanz mit Endzustand erklärt.

Eine Sprache heißt deterministisch kontextfrei, falls sie von einem DPDA K erkannt wird.

(Dazu gehören auch alle regulären Sprachen.)

Die Menge der deterministisch kontextfreien Sprachen $\mathcal{L}(K)$, die von einem PDA K erkannt werden, der die Bedingung $|\delta(q, x, \gamma)| + |\delta(q, \varepsilon, \gamma)| \leq 1$ erfüllt und mit leerem Keller aber nicht mit Endzustand akzeptiert, ist eine echte Teilmenge der deterministisch kontextfreien Sprachen.

Dazu gehört eine echte Teilmenge der regulären Sprachen, also nicht alle regulären Sprachen gehören dazu.

Die Menge der deterministisch kontextfreien Sprachen $\mathcal{L}(K)$ bildet eine echte Teilmenge der kontextfreien Sprachen und eine echte Obermenge der regulären Sprachen: $\mathcal{L}(\text{kontextfrei}) \supset \mathcal{L}(\text{det.kontextfrei}) \supset \mathcal{L}(\text{regulär})$

Damit gilt nicht wie bei endlichen Automaten, dass es für eine Sprache L zu jedem NPDA $K(L)$ einen äquivalenten DPDA $K'(L)$ gibt.

Z.B. gibt es für die Sprache $L = \{x \mid x = \text{Palindrom}\}$ nur NPDA's.

Für die Sprache $L = \{x \mid x = a^n b^n, n \geq 0\}$ gibt es dagegen sowohl NPDA's als auch DPDA's.

7.3 Beispiele

Kellerautomat 7.1: Gegeben ist die Sprache L :

$$L = \{x \mid x = a^n b^n, n > 0\}$$

Gesucht ist ein DPDA $K = (\Sigma, \Gamma, Q, \delta, \perp, q_0, F)$:

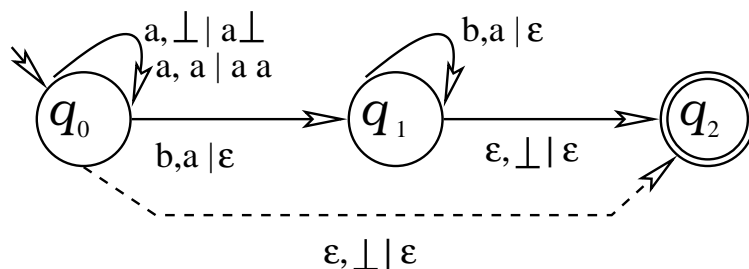


Bild 7.2: $K(L)$ mit $L = \{x \mid x = a^n b^n, n > 0\}$ (ohne - - - -)

$\Rightarrow Q = \{q_0, q_1, q_2\}$, mit $F = \{q_2\}$, $\Gamma = \{\perp, a, b\}$ und

$$\begin{aligned} \delta(q_0, a, \perp) &= (q_0, a \perp), & \delta(q_0, a, a) &= (q_0, a a), & \delta(q_0, b, a) &= (q_1, \varepsilon), \\ \delta(q_1, b, a) &= (q_1, \varepsilon), & \delta(q_1, \varepsilon, \perp) &= (q_2, \varepsilon). \end{aligned}$$

Für $L = \{x \mid x = a^n b^n, n \geq 0\}$ kann im Graphen $\delta(q_0, \varepsilon, \perp) = (q_2, \varepsilon)$ ergänzt werden, siehe (- - -). Der PDA ist dann nichtdeterministisch.

Deterministisch ist folgender PDA:

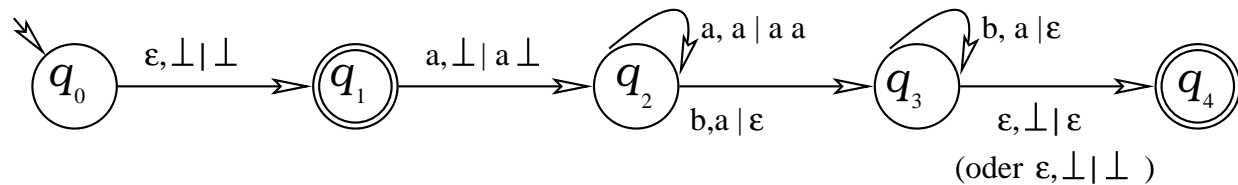


Bild 7.3: $K(L)$ mit $L = \{x \mid x = a^n b^n, n \geq 0\}$

Arbeitsweise des NPDA K : zu akzeptierendes Wort $x = a a a b b b$

Eingabeband	Keller
a a a b b b ↑ q_0	\perp ↑
a a a b b b ↑ q_0	a \perp ↑
a a a b b b ↑ q_0	a a \perp ↑
a a a b b b ↑ q_0	a a a \perp ↑
a a a b b b ↑ q_1	a a \perp ↑
a a a b b b ↑ q_1	a \perp ↑
a a a b b b ε ↑ q_1	\perp ↑
a a a b b b ε ↑ q_2	ε ↑

Kellerautomat 7.2: Entwerfen Sie einen NPDA K für die Sprache

$$L = \{x x^{\sim} \mid x \in \Sigma^*, x^{\sim} \text{ ist Spiegelwort von } x\}.$$

Problem: Mitte finden \Rightarrow 1. Mitte raten, 2. Prüfen, ob richtig geraten.

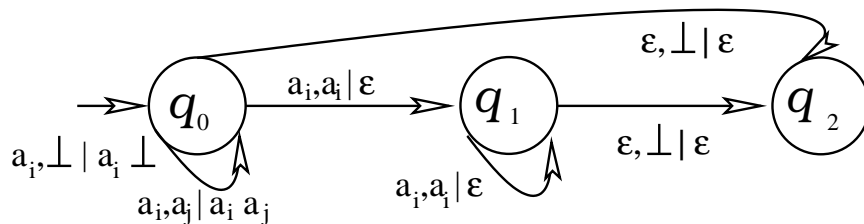


Bild 7.4: NPDA(, der mit leerem Keller akzeptiert)

Kellerautomat 7.3: Entwerfen Sie einen DPDA K für die Sprache

$$L = \{0^i 1 0^j \mid i \neq j, i, j \geq 0\}.$$

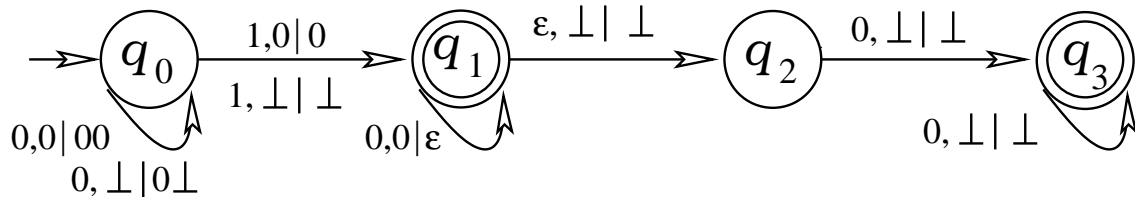


Bild 7.5: DPDA(, der mit Endzustand akzeptiert)

Entwerfen Sie einen NPDA K mit $L = L(K)$, der mit leerem Keller akzeptiert.

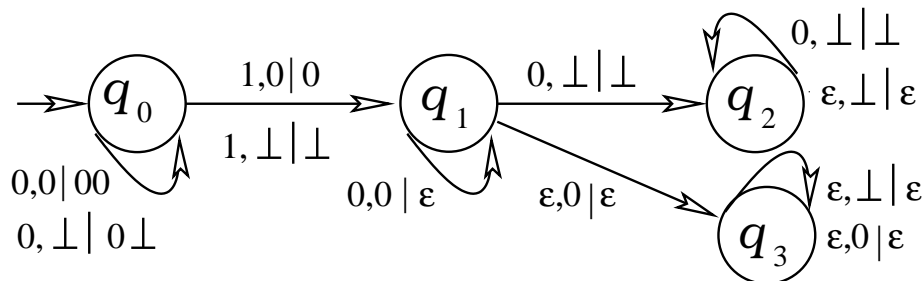


Bild 7.6: NPDA(, der mit leerem Keller akzeptiert)

$L = \{0^i 1 0^j \mid i \neq j, i, j \geq 0\}$ ist eine deterministisch kontextfreie Sprache.

7.4 Top-Down-Parsing, LL-Parsing

Wörter von Sprachen kontextfreier Grammatiken können mit Kellerautomaten verarbeitet werden, d.h. es kann festgestellt werden, ob das Wort von der Grammatik erzeugt werden kann. Dazu sind folgende Aktionen erlaubt:

1. **Expandieren E:** Ist das oberste Kellerzeichen eine Variable $A \in V$, ersetze dies durch eine „passende“ rechte Seite der Produktion(en) $A \rightarrow \dots$
2. **Lesen L:** Sind das oberste Kellerzeichen und das nächste Eingabezeichen dasselbe Zeichen $a \in \Sigma$, streiche a aus dem Keller und setze den Lesekopf auf dem Eingabeband ein Feld nach rechts.

Grammatik 7.1: Sprache $L = \{a^n b^m \mid n \geq 0, m \geq 0, \text{gerade}\}$

$V = \{S, A, B\}$, (S ist das Satzsymbol), $\Sigma = \{a, b\}$,

P: $(S, AB)_1, (A, aA)_2, (A, \varepsilon)_3, (B, Bbb)_4, (B, \varepsilon)_5$

\Rightarrow Kelleralphabet $\Gamma = \{S, A, B, a, b\}$, Kellernullzeichen $\perp = S$

gelesen	Resteingabe	Keller	Aktion
	a a a b b b b	S	Start, E (1)
	a a a b b b b	AB	E (2)
	a a a b b b b	a AB	L
a	a a b b b b	AB	E (2)
a	a a b b b b	a AB	L
aa	a b b b b	AB	E (2)
aa	a b b b b	a AB	L
aaa	b b b b	AB	E (3)
aaa	b b b b	B	E (4)
aaa	b b b b	Bbb	E (4)
aaa	b b b b	Bbbbb	E (5)
aaa	b b b b	bbbbb	L
aaab	b b b	bbb	L
aaabb	b b	bb	L
aaabbb	b	b	L
aaabbbb	ε	ε	Schluss

Betrachtet man für die Expansionsschritte zeilenweise gelesene Buchstaben verkettet mit dem Kellerinhalt, dann ergibt sich von oben nach unten gelesen eine Linksableitung für das Eingabewort.

Das Lesen eines Wortes $w \in L(G)$ unter Erzeugung einer Ableitung nennt man **Parsing**. Wird das Wort von **links** nach rechts gelesen und wird dabei eine **Linksableitung** erzeugt, dann nennt man das **LL-Parsing**.

Setzt man die Linksableitung in einen entsprechenden Ableitungsbaum um, dann wird dieser von der Wurzel zu den Blättern hin entwickelt. Deshalb nennt man LL-Parsing auch **Top-Down-Parsing**.

Satz 7.3: Zu jeder kontextfreien Grammatik G gibt es einen NPDA M mit $L(G) = L_M$. Der Kellerautomat hat nur einen Zustand.

Formal sieht M so aus:

$M = (Q, \Sigma, \Gamma, q_0, \perp, \delta)$ mit $Q = \{q_0\}$, $\Gamma = V \cup \Sigma$, $\perp = S$,

$\delta(q_0, a, a) = \{(q_0, \varepsilon)\}$, $a \in \Sigma$, (Leseschritt)

$\delta(q_0, \varepsilon, A) = \{(q_0, \gamma) \mid A \rightarrow \gamma \text{ ist in } P\}$, $A \in V$, (Expansionsschritt).

Für eindeutige Grammatiken ist das Verfahren deterministisch, ansonsten nichtdeterministisch.

Um einen Parser zu erhalten, der genau nach dem Top-Down-Verfahren arbeitet, kann man, falls möglich, die Grammatik in eine LL(1)-Grammatik umwandeln („lookahead“ 1). Eine LL(1)-Grammatik zwingt den Parser, anhand des aktuellen Eingabebuchstaben die richtige Produktion zu wählen.

Ist das nicht möglich, kann man mit LL(k)-Grammatiken für beliebige $k \geq 1$ LL(k)-Parser entwerfen. Der LL(k)-Parser betrachtet die nächsten k Zeichen und wählt danach die passende Produktion. Näheres Compilerbau.

7.5 Bottom-Up-Parsing, LR-Parsing

Ein anderer Ansatz, wie Wörter zu kontextfreien Grammatiken G mit Kellerautomaten verarbeitet werden können, ist das LR-Parsing. Die Eingabewörter werden wieder von **links** nach rechts gelesen, aber es wird eine **Rechtsableitung** erzeugt. Der Ableitungsbaum entsteht von den Blättern zur Wurzel. Im Keller befindet sich das Kellerendezeichen \perp und es sind folgende Aktionen erlaubt:

1. **shift:** In den Keller wird das nächste Eingabezeichen gelesen.
2. **reduce:** Falls die s oberen Kellerzeichen die rechte Seite einer Produktion $A \rightarrow \gamma$ mit $|\gamma| = s$ bilden, wird γ entfernt und A wird in den Keller geschrieben.

Grammatik 7.1: Sprache $L = \{a^n b^m \mid n \geq 0, m \geq 0, \text{ gerade}\}$

$V = \{S, A, B\}$, (S ist das Satzsymbol), $\Sigma = \{a, b\}$,

P: $(S, AB)_1$, $(A, aA)_2$, $(A, \varepsilon)_3$, $(B, Bbb)_4$, $(B, \varepsilon)_5$

\Rightarrow Kelleralphabet $\Gamma = \{S, A, B, a, b\}$, Kellerranfanzzeichen \perp

Aktion	Keller	Eingabe	Bem.	Rechtsableitung, \uparrow , s.u.
	\perp	a a a b b b b		
shift	\perp <u>a</u>	a a a b b b b	shift	
shift	\perp a a	a a a b b b b	shift	
shift	\perp a a a <u>ε</u>	a a a b b b b	reduce	\Rightarrow a a a b b b b
reduce (3)	\perp a a a <u>A</u>	a a a b b b b	reduce	\Rightarrow a a a A b b b b
reduce (2)	\perp a a <u>A</u>	a a a b b b b	reduce	\Rightarrow a a A b b b b
reduce (2)	\perp <u>a</u> A	a a a b b b b	reduce	\Rightarrow a A b b b b
reduce (2)	\perp A <u>ε</u>	a a a b b b b	reduce	\Rightarrow A b b b b
reduce (5)	\perp A B	a a a b b b b	shift	
shift	\perp A B b	a a a b b b b		
shift	\perp A <u>B</u> b b	a a a b b b b	reduce	\Rightarrow A B b b b b
reduce(4)	\perp A B	a a a b b b b	shift	
shift	\perp A B b	a a a b b b b		
shift	\perp A <u>B</u> b b	a a a b b b b	reduce	\Rightarrow A B b b
reduce(4)	\perp <u>A</u> B	a a a b b b b	reduce	$S \Rightarrow AB$
reduce(1)	\perp S	a a a b b b b		
Keller leeren	\perp	a a a b b b b		
	ε	a a a b b b b		

Betrachtet man für die Reduceschritte zeilenweise Kellerinhalt verkettet mit der Resteingabe, dann ergibt sich von unten nach oben gelesen eine Rechtsableitung für das Eingabewort w .

Um s Zeichen des Kellers lesen zu können, gibt es technische Mittel, die hier nicht weiter erläutert werden. Es kann während der Abarbeitung zu „shift-reduce-“ oder „reduce-reduce-“Konflikten kommen. Dazu gibt es wieder eine Theorie, wie diese behandelt bzw. vermieden werden können. Das führt wieder zu LR(1)- oder LR(k)-Grammatiken bzw. LR(1)- oder LR(k)-Parsern.

Kapitel 8

TURING-Maschinen und Berechenbarkeit

Es gibt auch jeweils Klassen von „Maschinen“, die Sprachen vom Typ 0 bzw. Typ 1 (\mathcal{L}_0 und \mathcal{L}_1) erkennen können.

Für die kontextsensitiven Sprachen (\mathcal{L}_1) ist dies der **linear beschränkte Automat** und für die Typ-0-Sprachen (\mathcal{L}_0) die nachfolgend behandelte **TURING-Maschine**.

8.1 Algorithmus und Berechenbarkeit - eine Einführung

Der Begriff des Algorithmus ist mit dem der Berechenbarkeit äußerst eng verknüpft. Nur mittels einer schematischen Rechenvorschrift (Algorithmus) kann die zugehörige Funktion berechnet werden.

So kann man zum Beispiel eine evtl. partielle Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ als berechenbar ansehen, falls es einen Algorithmus gibt, der mit $(n_1, n_2, \dots, n_k) \in \mathbb{N}^k$ gestartet, nach endlich vielen Schritten mit der Ausgabe von $f(n_1, n_2, \dots, n_k)$ hält.

Ist die Funktion partiell, so möge der Algorithmus an den nicht definierten Stellen nicht halten.

Frage: gibt es Funktionen, die nicht berechenbar sind?

Antwort „Ja“, und hierfür gibt es einen ganz einfachen Beweis, der anhand eines Beispiels erläutert werden soll:

Beispiel 8.1: Man beweise, dass die Menge $Abb(\mathbb{N}, \{0, 1\})$ nichtberechenbare Funktionen enthält:

1. die Menge $Abb(\mathbb{N}, \{0, 1\})$ ist überabzählbar.

Annahme, die Menge $Abb(\mathbb{N}, \{0, 1\})$ ist abzählbar:

	1	2	3	4	5	6	7	...	j	...
f_1	$f_1(1)$	$f_1(2)$	$f_1(3)$	$f_1(4)$	$f_1(5)$	$f_1(6)$	$f_1(j)$...
f_2	$f_2(1)$	$f_2(2)$	$f_2(3)$	$f_2(4)$	$f_2(5)$	$f_2(6)$...		\vdots	
f_3	$f_3(1)$	$f_3(2)$	$f_3(3)$	$f_3(4)$...					
f_4	$f_4(1)$	$f_4(2)$	$f_4(3)$...						
f_5	$f_5(1)$	$f_5(2)$...							
\vdots	\vdots								\vdots	
f_i	$f_i(1)$		$f_i(j) \in \{0, 1\}$...
\vdots	\vdots								\vdots	

Entwerfe eine Funktion $g : \mathbb{N} \rightarrow \{0, 1\}$ mit $g(i) = 0 \Leftrightarrow f_i(i) = 1$ für $i = 1, 2, 3 \dots j \dots$.

g müsste in $f_1 \dots f_j \dots$ enthalten sein, dh. $\exists k \in \mathbb{N}$, für das $g = f_k$, insbesondere $g(k) = f_k(k)$ gilt

es gilt aber $g(k) = 0 \Leftrightarrow f_k(k) = 1 \Rightarrow$ Widerspruch

\Rightarrow die Menge $Abb(\mathbb{N}, \{0, 1\})$ ist überabzählbar.

2. die Menge der berechenbaren Funktionen in $Abb(\mathbb{N}, \{0, 1\})$ ist abzählbar:

f sei eine beliebige berechenbare Funktion.

$f \rightarrow A_f$, dh. man kann für eine berechenbare Funktion f einen Algorithmus A_f angeben.

A_f ist ein Programm, dem eine Alphabet Σ zugrunde liegt, über dem man A_f als Wort angeben kann, dh. $A_f \in \Sigma^*$.

Σ^* (Menge aller Wörter über Σ) ist abzählbar.

Da $\{A_f\} \subset \Sigma^*$ ist, ist auch $\{A_f\}$ abzählbar.

$\{A_f(\mathbb{N} \rightarrow \{0, 1\})\} \subset \{A_f\}$, damit ist $\{A_f(\mathbb{N} \rightarrow \{0, 1\})\}$ abzählbar.

Da die Menge $Abb(\mathbb{N}, \{0, 1\})$ überabzählbar ist aber

$\{A_f(\mathbb{N} \rightarrow \{0, 1\})\}$ abzählbar, gibt es nichtberechenbare Funktionen.

8.2 Die TURING-Maschine und -Berechenbarkeit

Eine Formalisierung des Algorithmusbegriffs bzw. der Berechenbarkeit gelang 1936 Alan TURING (1912-1954, englischer Mathematiker, Kryptoanalytiker und Computerkonstrukteur) mit dem nach ihm benannten Modell der TURING-Maschine.

Dieses Modell besteht aus einem potentiell unendlichen Band, das in Felder unterteilt ist.

Jedes Feld kann ein einzelnes Zeichen des Arbeitsalphabets aufnehmen.

Ein Schreib-Lese-Kopf liest in jedem Schritt das an der aktuellen Stelle befindliche Zeichen, kann dieses durch ein anderes überschreiben und sich um maximal eine Position nach links oder rechts bewegen.

Zum Arbeitsalphabet gehört ein spezielles „Blank“-Zeichen B .

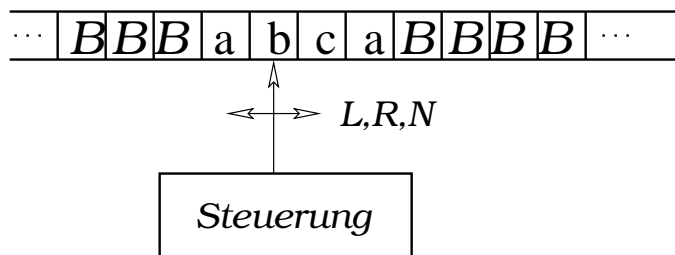


Bild 8.1: allgemeine Turingmaschine

Def. 8.1: Eine deterministische Turing-Maschine M ist ein Tupel

$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ mit

Q endliche Menge von Zuständen,

Σ Eingabealphabet,

Γ Arbeitsalphabet, $\Gamma \supset \Sigma$,

δ Zustandsüberföhrungsfunktion mit

$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, N\} \cup \{\perp\}$, wenn $\delta(q, a) = \perp$, dann gilt $\delta(q, a)$ als undefiniert, d.h. δ ist eine partielle Funktion

$q_0 \in Q$ Initial- oder Startzustand,

B Blank-Zeichen mit $B \in \Gamma - \Sigma$,

$F \subseteq Q$ Menge von Final- oder Endzuständen.

Dabei bedeutet $\delta(q, a) = (q', a', b)$ folgendes:

- Beim Lesen des Zeichens a vom Band im Zustand q geht die TM M in den Zustand q' über,
- ersetzt das gelesene Zeichen a auf dem Band durch das Zeichen a' und
- verschiebt den Kopf entsprechend b (L =links, R =rechts, N =nicht bewegen).

Eine „Momentaufnahme“ einer TURING-Maschine nennen wir Konfiguration:

Def. 8.2: Als **Konfiguration** einer Turing-Maschine M legen wir das Tupel $(u, q, v) \in \Gamma^* \times Q \times \Gamma^+$ fest mit $v = B$ oder $v = av_1$ $a \in \Sigma$ und $a \neq B$. Der LS-Kopf zeigt auf das erste Symbol von v . Die Konfiguration wird i.Allg. durch ein Wort $k = u q v \in \Gamma^* \times Q \times \Gamma^+$ dargestellt.

Def. 8.3: Die durch einen **Schritt** der TM M aus der Konfiguration

$k = a_1 \dots a_{i-2} a_{i-1} q a_i a_{i+1} \dots a_n, \forall i \in \{1, \dots, n\} : (a_i \in \Gamma)$ hervorgehende Folgekonfiguration $k', k \vdash_M k'$, erhalten wir wie folgt:

- $k \vdash_M a_1 \dots a_{i-2} q' a_{i-1} a' a_{i+1} \dots a_n$, falls $\delta(q, a_i) = (q', a', L)$ und $i > 1$;
- $k \vdash_M a_1 \dots a_{i-2} a_{i-1} a' q' a_{i+1} \dots a_n$, falls $\delta(q, a_i) = (q', a', R)$ und $i > 1$;
- $k \vdash_M a_1 \dots a_{i-2} a_{i-1} q' a' a_{i+1} \dots a_n$, falls $\delta(q, a_i) = (q', a', N)$ und $i > 1$;

Folgende Sonderfälle müssen noch behandelt werden:

- $q a_1 a_2 \dots a_n \vdash_M q' B a' a_2 \dots a_n$, falls $\delta(q, a_1) = (q', a', L)$;
- $a_1 a_2 \dots a_{n-1} q a_n \vdash_M a_1 a_2 \dots a_{n-1} a' q' B$, falls $\delta(q, a_n) = (q', a', R)$.

Die reflexive transitive Hülle von \vdash_M schreiben wir als: \vdash_M^* .

($k \vdash_M^* k'$ besagt, dass sich eine Konfiguration k' aus einer Konfiguration k durch eine endliche Folge von Schritten ableitet.)

Def. 8.4: Die von einer TM M **akzeptierte Sprache** L_M ist folgendermaßen definiert:

$$L_M = \{x \in \Sigma^* \mid q_0 x \vdash_M^* u q v \text{ mit } u \in \Gamma^*, v \in \Gamma^+, q \in F\}.$$

Die Klasse der von TURING-Maschinen erkennbaren Sprachen ist die Klasse der rekursiv aufzählbaren Sprachen. In dieser Klasse ist die der regulären Sprachen echt enthalten.

Def. 8.5: Eine partielle Wortabbildung $f : \Sigma^* \rightarrow \Delta^*$ heißt **turing-berechenbar**, falls es eine deterministische Turing-Maschine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ mit $\Delta \subseteq \Gamma$ gibt, so dass

- a) $H_M = D(f)$, Haltemenge der TM M
- b) $q_0 x \vdash_M^* q f(x)$ mit $q \in F$ für alle $x \in H_M$.

Bemerkung:

Zur Berechnung einer partiellen Funktion $f : \mathbb{N}^k \rightarrow \mathbb{N}$ sind $\Sigma = \{0, 1\}$ und $\Delta = \{0, 1\}$ geeignet und das Lesen der Worte als Zahlen.

Binär-Darstellung oder Unäre Darstellung der Zahlen. Bei der unären Darstellung wird die ganze Zahl n dabei durch eine Zeichenkette $\mathbf{1}^{n+1}$ dargestellt, die wir als $un(n)$ bezeichnen wollen.

Unter dem Aspekt der Verknüpfung von TM'en ist es wichtig, dass der Kopf am Ende der Berechnung **unter dem ersten Zeichen** vom Ergebnis auf dem Band steht. (Siehe Def. **turing-berechenbar**).

Man kann zeigen, dass auch eine mehrbändige TURING-Maschine, die auf k Bändern mit k unabhängigen Köpfen gleichzeitig jeweils wie eine normale TM arbeitet, nicht mehr leistet als die einbändige TM.

Def. 8.6: Eine **deterministische Mehrband-TURING-Maschine** mit k Bändern, kurz k -Bd.TM, ist ein Tupel $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$; dabei sind Q, Σ, Γ, q_0 und F wie bei der einbändigen TM festgelegt, die Zustandsüberföhrungsfunktion jedoch als $\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R, N\}^k$.

Satz 8.1: Zu jeder k -Bd.TM M gibt es eine (Einband-)TM M' mit $L_M = L_{M'}$ bzw. so, dass M' dieselbe Funktion wie M berechnet.

Def. 8.7: Eine **nichtdeterministische Turing-Maschine**, kurz NTM, ist eine Einband-TM mit endlicher Kontrolle und einer nichtdeterministischen Zustandsüberföhrungsfunktion: $\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R, N\})$.

Die NTM akzeptiert ihre Eingabe, wenn eine Folge möglicher Schritte in einen Endzustand führt.

Die Bezeichnung f_M (, d.h. M berechnet eine Funktion f ,) ist für eine nichtdeterministische TM M nicht definiert, darf also nicht benutzt werden.

Satz 8.2: Zu jeder NTM M gibt es eine deterministische TM M' mit $L_M = L_{M'}$.

8.3 Beispiele

Beispiel 8.2: TM zur Berechnung von $f(n) = n + 1$, $n \geq 0$, unär.
 Der LS-Kopf steht auf der ersten linken 1.

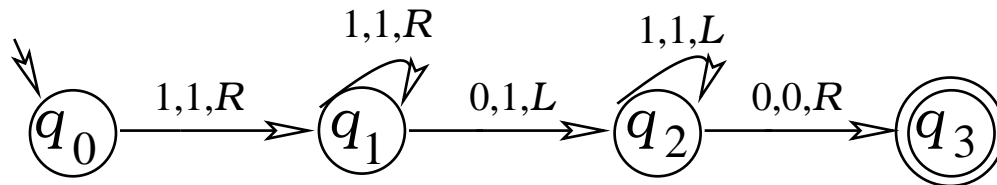


Bild 8.2: TM $M(f(n) = n + 1)$, unäre Darstellung

Beispiel 8.3: TM zur Berechnung von $f(n) = n + 1$, $n \geq 0$, binär.
 Der LS-Kopf steht auf dem ersten Zeichen von links.

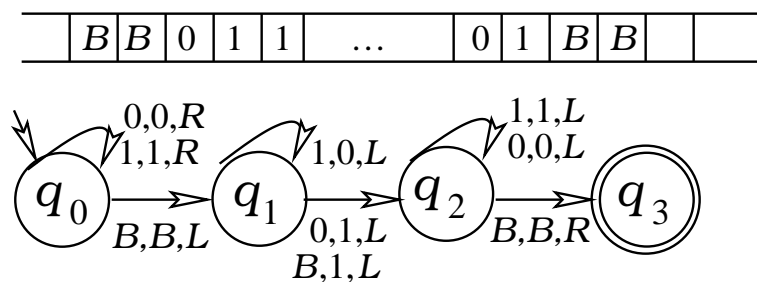


Bild 8.3: TM $M(f(n) = n + 1)$ binäre Darstellung

Beispiel 8.4: TM zur Berechnung von $f(n, a) = n + a$, $n, a \geq 0$, unär.
 Der LS-Kopf steht auf der ersten linken 1.

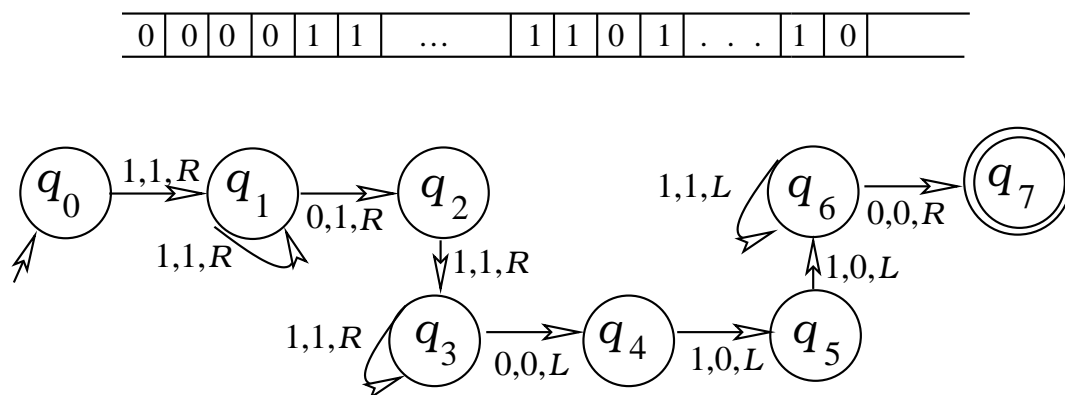


Bild 8.4: TM $M(f(n, a) = n + a)$ unäre Darstellung

Beispiel 8.5: 2-Bd.TM zur Berechnung von $f(n) = 2n, n \geq 0$, unär

Die Eingabe steht auf Bd.1, die Ausgabe soll auf Bd.1 stehen.

Der LS-Kopf steht auf der ersten linken 1 auf Bd.1 und auf einer beliebigen 0 auf Bd.2.

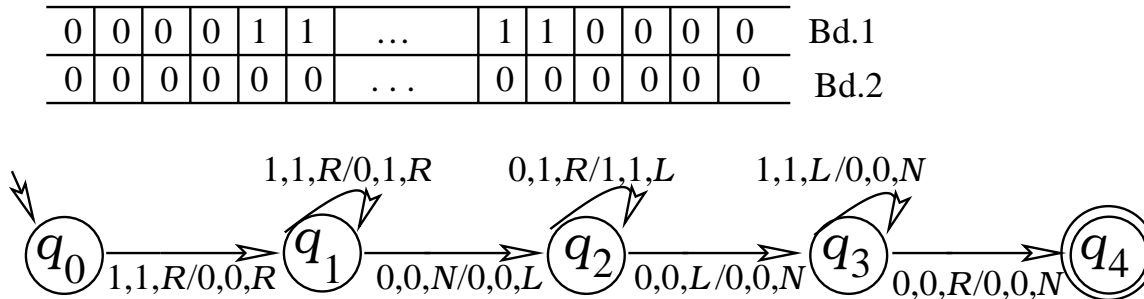


Bild 8.5: TM $M(f(n) = 2n)$

Beispiel 8.6: Entwurf einer TM zur Akzeptanz der Sprache

$$L = \{a^n b^n c^n \mid n \geq 1\}. \Gamma = \{a, b, c, x, B\}$$

Der LSK steht auf dem ersten linken Zeichen vom Eingabewort.

Das Band soll nach der Abarbeitung leer sein.

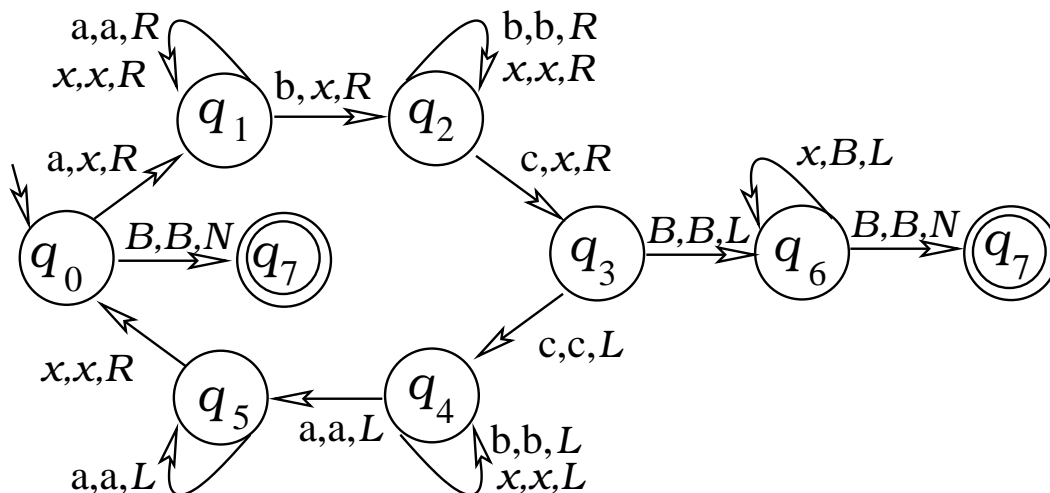


Bild 8.6: TM, die $L = \{a^n b^n c^n \mid n \geq 0\}$ akzeptiert.

Beispiel 8.7: TM für **busy-beaver-Fkt. $bb(n)$** (/BRAU90/) **weglassen**

Fleißiger Biber: Holz für Dammbau suchen, fällen, zum Damm bringen, bauen, gesucht fleißigster Biber \Rightarrow Algorithmus auf TM bringen und untersuchen.

TM: Band initial leer, d.h. mit Nullen belegt, für jeden Baum, der gefällt wird und verbaut wird, wird eine 1 geschrieben.

Unter $bb(n)$ versteht man die maximale Anzahl von 1'en, die eine haltende TM mit n Zuständen hintereinander auf das initial leere Band schreibt.

Falls TM nicht hält, ist $bb(n) = 0 \Rightarrow bb : \mathbb{N} \rightarrow \mathbb{N}$

Für $n = 2$ mögliche Variante:

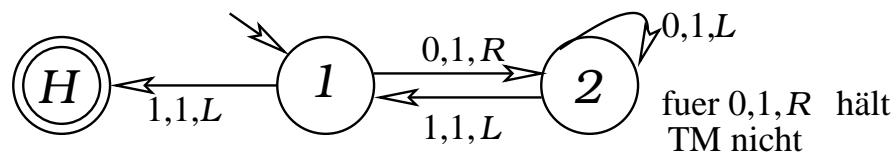


Bild 8.7: mögliche Turingmaschine für $bb(2)$

bb -Programm mit 2 Anweisungen:

- Zst 1: falls $x = 0$ schreibe 1, LSK nach R , TM in Zst 2
- $x = 1$ 1, LSK nach L , TM in Zst Halt
- Zst 2: falls $x = 0$ schreibe 1, LSK nach L , TM in Zst 2
- mit R hält TM nicht \rightarrow Abbruch vereinbaren
- $x = 1$ 1, LSK nach L , TM in Zst 1

Nach 6 Schritten sind 3 Striche erzeugt.

Das Band der Turingmaschine ist initial mit Nullen belegt



Berechnung der Anzahl der Striche $bb(n)$:

- Man schreibe alle bb -Programme für n -Zustände, das sind $(4(n + 1))^{2n}$, z.B. für $n = 6$ sind das mehr als 2×10^{17} .
- Man prüfe jedes Programm ob es **hält** und wenn ja, **wieviele Striche** erzeugt wurden.
- $bb(n)$ ist Maximum der Striche \Rightarrow für $n = 6$ weniger als 2075 bei mehr als 4 208 824 Schritten. (Schult 1982 in /BRAU90/)

Es hat sich gezeigt, dass die bb -Fkt $bb(n)$ unberechenbar ist, d.h.

es gibt kein allgemeines Verfahren - kein Rechnerprogramm - kein Programm für TM'en - mit dem man den Wert $bb(n)$ für jede natürliche Zahl n ausrechnen kann. (Beweis in Brauer: Grenzen maschineller Berechenbarkeit, Informatik-Spektrum (1990) 13:61-70, Springer Verlag)

Grund 1 ist die Definition von $bb(n)$, und zwar der Teil, dass zuerst festgestellt werden muss, ob das bb -Programm hält.

Es gibt kein allgemeines Verfahren, mit dem in allen Fällen festgestellt werden kann, ob ein bb -Programm hält oder nicht.

Diese Aussage gilt für alle Programme, nicht nur bb -Programme (z.B. Pascal-Programme).

Das Halteproblem für diese Problemklasse ist nicht entscheidbar.

\Rightarrow praktische Konsequenz:

Das Betriebssystem eines Computers ist nicht in der Lage zu entscheiden, ob

- ein Programm (, das auf dem Computer läuft,) jemals hält (und ein Ergebnis liefert) oder
- ein Programmfehler vorliegt, so dass es nie hält.

\Rightarrow Frist für Abbruch setzen.

Grund 2: Alle $(4(n + 1))^{2n}$ Programme erstellen:

n	Anzahl
1	64
2	20736
3	16 777 216
4	$2,56 \times 10^{10}$
5	$> 6,3 \times 10^{13}$
6	$> 2 \times 10^{17}$

Die Welt besteht bisher $\sim 5 \times 10^{17}$ Sekunden,

Anzahl der Programme $\sim 2 \times 10^{17}$

$\Rightarrow bb(n)$ für $n = 5$ bisher nicht berechnet, nur Abschätzung, Grund liegt im Berechnungsaufwand:

- Anzahl der Programme
- Anzahl der Schritte > 2 Million
- Algorithmus für Aussortieren nichthaltender Programme
- ⋮

$\Rightarrow bb(10)$ ist praktisch unberechenbar.

Betrachtungen zum Aufwand, z.B. was können Computer, wenn Speicherplatz und Zeit beschränkt sind, untersucht man im Rahmen der **Komplexitätstheorie**.

Was eine Turing-Maschine leistet, scheint nun eher bescheiden zu sein.

Tatsächlich hat man aber noch kein mächtigeres Modell gefunden.

Andere Modelle für Algorithmen bzw. Berechnungen (s.u.) erfassen auch nur genau dieselbe Klasse von Funktionen.

Bekannt als CHURCHsche These ist folgende Schlussfolgerung:

These: Die durch die formale Definition der TURING-Berechenbarkeit erfasste Klasse von Funktionen stimmt genau mit der Klasse der intuitiv berechenbaren Funktionen überein.

Begreiflicherweise ist diese These nicht beweisbar, solange der intuitive Berechenbarkeitsbegriff nicht formal definiert ist.

Sehr wohl beweisbar ist jedoch die Äquivalenz der bisher gefundenen formalen Berechenbarkeitsmodelle, was die obige These sehr erhärtet.

Offenbar muss es gelingen, jede noch so komplizierte Funktion, wenn sie denn überhaupt berechenbar ist, auf einer TURING-Maschine auszuführen.

Insbesondere gilt es auch nachzuweisen, dass selbst alle modernen digitalen Computer nicht mehr als eine TURING-Maschine leisten.

Kapitel 9

Aufzählbarkeit, Entscheidbarkeit, rekursive Sprachen

9.1 Aufzählbarkeit und Entscheidbarkeit

Der Begriff der Berechenbarkeit ist vor allem auf Funktionen zugeschnitten.

Bei Sprachen ging es um das Problem der Erkennbarkeit, d.h. zu entscheiden, ob eine gegebene Zeichenkette Wort einer bestimmten Sprache ist.

Dazu kann man folgende Korrespondenz herstellen:

Def. 9.1: Eine Sprache $L \subseteq \Sigma^*$ heißt **entscheidbar**, falls ihre charakteristische Funktion $\chi_L : \Sigma^* \rightarrow \{0, 1\}$ berechenbar ist für alle $x \in \Sigma^*$:

$$\chi_L(x) = \begin{cases} 1, & \text{falls } x \in L \\ 0, & \text{falls } x \notin L \end{cases}$$

Bemerkung:

Da die charakteristische Funktion total ist, muss eine sie berechnende TM M in jedem Fall stoppen. Ist M dabei in einem Endzustand, so ist die Eingabe offenbar akzeptiert.

Stoppt M in einem anderen Zustand, so akzeptiert M die Eingabe nicht.

Def. 9.2: Eine Sprache $L \subseteq \Sigma^*$ heißt **semi-entscheidbar**, falls ihre „halbe“ charakteristische Funktion $\chi_L^+ : \Sigma^* \rightarrow \{1\}$ berechenbar ist.

Es gilt für alle $x \in \Sigma^*$

$$\chi_L^+(x) = \begin{cases} 1 & , \quad \text{falls } x \in L \\ \text{undefiniert,} & \text{sonst} \end{cases}$$

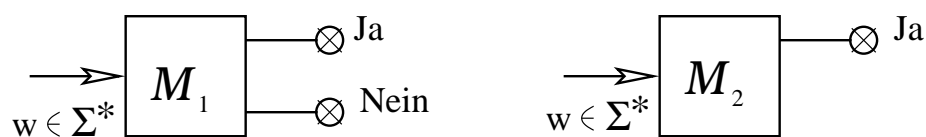
Bemerkung:

Der Unterschied zwischen beiden Definitionen besteht darin, dass bei einer entscheidbaren Sprache für jedes Wort nach endlich vielen Schritten feststeht, ob es Element der Sprache ist oder nicht.

Bei einer semi-entscheidbaren Sprache steht für jedes in der Sprache enthaltene Wort ebenfalls nach endlich vielen Schritten die Zugehörigkeit fest.

Wenn jedoch nach einer beliebigen Anzahl von Schritten die Zugehörigkeit noch nicht feststeht, so ist nicht klar, ob das Wort nicht zur Sprache gehört oder nur noch nicht genügend Schritte ausgeführt wurden, das heißt, die Nichtzugehörigkeit ist nicht entscheidbar.

Folgende Schemata sollen das verdeutlichen (links eine TM für eine entscheidbare, rechts für eine semi-entscheidbare Sprache):



Offensichtlich gilt damit:

Satz 9.1: Eine Sprache L ist entscheidbar genau dann, wenn sowohl L als auch deren Komplement \bar{L} semi-entscheidbar sind.

Die von TM'en erkennbaren (nicht: entscheidbaren!) Sprachen bilden die Klasse der rekursiv aufzählbaren Sprachen. Dieser Begriff soll im folgenden definiert und mit dem der Semi-Entscheidbarkeit verglichen werden.

Def. 9.3: Eine Sprache $L \subseteq \Sigma^*$ heißt **rekursiv aufzählbar**, falls entweder $L = \emptyset$ oder es existiert eine totale berechenbare Funktion $f : \mathbb{N} \rightarrow L$, so dass $L = \{f(0), f(1), f(2), \dots\}$.

(„ f zählt L auf“. Hierbei ist offensichtlich $f(i) = f(j)$ für $i \neq j$ zulässig.)

Jede Teilmenge einer abzählbaren Menge ist wieder eine abzählbare Menge, (hingegen muss eine Teilmenge einer rekursiv aufzählbaren Sprache keineswegs ebenfalls rekursiv aufzählbar sein. Σ^* ist rekursiv aufzählbar, aber bei weitem nicht jede Sprache $L \subseteq \Sigma^*$!).

Satz 9.2: Eine Sprache L ist rekursiv aufzählbar genau dann, wenn sie semi-entscheidbar ist.

Beweis:

$L \subseteq \Sigma^*$ heißt rekursiv aufzählbar, falls entweder $L = \emptyset$ oder es existiert eine totale berechenbare Funktion $f : \mathbb{N} \rightarrow L$, so dass $L = \{f(0), f(1), f(2), \dots\}$.

$L \subseteq \Sigma^*$ ist semi-entscheidbar heißt, \exists eine TM M_{χ^+} , die $\chi_L^+(x)$ berechnet.

\Rightarrow Sei $f : \mathbb{N} \rightarrow L$ eine Aufzählung der Sprache L .

Für jedes $x \in \Sigma^*$ werden die Wörter $f(n)$ der Sprache L für $n = 0, 1, 2, 3, \dots$ aufgezählt und $x \in \Sigma^*$ wird als Eingabe einer TM M genutzt, die feststellt, ob $x \in \Sigma^* = f(\mathbb{N})$.

Für alle $f(n) = x \in \Sigma^*$ wird die TM M nach endlich vielen Schritten das Ergebnis $\chi_L^+(x) = 1$ liefern. Ansonsten hält sie nicht an.

\Leftarrow L sei semi-entscheidbar.

Wir brauchen eine TM G , die alle Wörter von Σ^* mit wachsender Länge und innerhalb der gleichen Länge lexikographisch generiert.

Es ist jedoch nicht möglich, einfach die erkennende TM M_L auf die in dieser Reihenfolge erzeugten Zeichenketten anzusetzen, da sie auf Wörtern, die in der Sprache nicht enthalten sind, nicht hält.

Dieses Problem ist aber mit einer Schrittzahlbegrenzung lösbar (Dove-tailing):

Ist ein $x_k \in \Sigma^*$ in $L(M)$, so akzeptiert M_L gemäß Voraussetzung nach einer endlichen Schrittzahl s .

Wenn nun eine TM konstruierbar ist, die sämtliche Paare $(n, m) \in \mathbb{N}^2$ in einer gewissen Reihenfolge erzeugt, so muss auch (k, s) ($\hat{=}$ x_k wird von M_L nach s Schritten akzeptiert) nach endlich vielen Schritten darunter sein. Wird für jedes Paar (n, m) indessen die Zeichenkette x_n generiert und die TM M darauf m Schritte angesetzt, so wird bei $n = k$ und $m = s$ das Wort x_k offenbar als zur Sprache gehörig identifiziert und kann somit ausgegeben (= aufgezählt) werden.

Hält die Erkennung nach dem m -ten Schritt nicht, so wird das nächste Paar (n, m) erzeugt.

\mathbb{N}^2 ist rekursiv aufzählbar mit dem CANTORschen Diagonalisierungsverfahren.

Bei Eingabe einer Zahl i wird in endlicher Zeit das entsprechende (n, m) -Paar erzeugt, anschließend in endlicher Zeit das zugehörige Wort $x_n \in \Sigma^*$ generiert und mit m Schritten zu akzeptieren versucht. Falls das Wort

erkannt ist, wird es ausgegeben. Mit einer Schleife kann somit die Sprache rekursiv aufgezählt werden, wenn sie semi-entscheidbar ist. W.z.b.w.

Satz 9.3: Eine Sprache $L \subseteq \Sigma^*$ ist genau dann rekursiv aufzählbar, wenn sie vom CHOMSKY-Typ 0 ist (d.h. durch eine allgemeine Grammatik erzeugt werden kann).

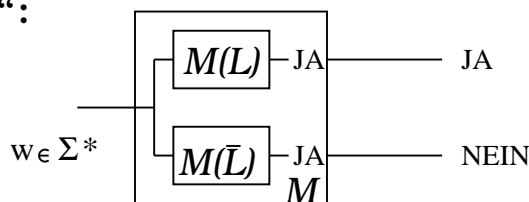
Bemerkung: Satz besagt die Äquivalenz von allgemeiner Grammatik und Turingmaschine.

9.2 Rekursive Sprachen

Def. 9.4: Eine Sprache $L \subseteq \Sigma^*$ heißt **rekursiv**, falls sie von wenigstens einer TM erkannt wird, die auf jeder Eingabe hält. (Dabei muss das Anhalten nicht mit Akzeptieren verbunden sein.)

Satz 9.4: Eine Sprache $L \subseteq \Sigma^*$ ist genau dann rekursiv, wenn sowohl L als auch \bar{L} rekursiv aufzählbar sind.

„Beweis“:

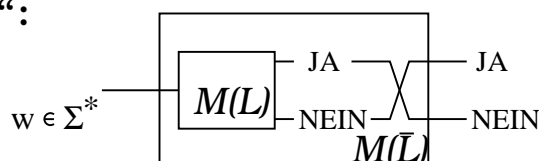


Satz 9.5: Eine Sprache ist genau dann rekursiv, wenn sie entscheidbar ist.

Rekursive Sprachen haben folgende Eigenschaften:

Satz 9.6: Abgeschlossenheit bezüglich Komplement: Das Komplement einer rekursiven Sprache ist eine rekursive Sprache.

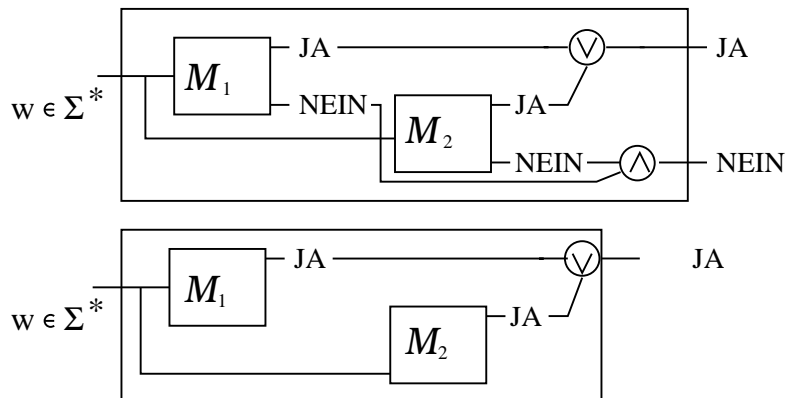
„Beweis“:



Satz 9.7: Abgeschlossenheit bezüglich Vereinigung:

Die Vereinigung zweier rekursiver Sprachen ist rekursiv. Die Vereinigung zweier rekursiv aufzählbarer Sprachen ist rekursiv aufzählbar.

„Beweis“:



Für eine Sprache L und ihr Komplement \bar{L} ist genau einer der folgenden drei Fälle möglich:

- Weder L noch \bar{L} sind rekursiv aufzählbar.
- Entweder L oder \bar{L} ist rekursiv aufzählbar, die komplementäre Sprache \bar{L} bzw. L aber nicht. Dann ist natürlich weder L noch \bar{L} rekursiv.
- Sowohl L als auch \bar{L} sind rekursiv aufzählbar, somit sind auch beide rekursiv.

9.3 Unentscheidbarkeit und Halteproblem

9.3.1 Kodierung von TURING-Maschinen

Häufig ist es notwendig, Aussagen über die Entscheidbarkeit von Problemen zu treffen. Probleme werden dabei als Wörter über einem Alphabet formuliert.

Da auch Aussagen über TURING-Maschinen selbst gemacht werden sollen, müssen wir eine Codierung von TURING-Maschinen angeben.

Beginnen wir mit einer normierten TM M :

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F) = (Q, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\}).$$

über eine binäre Codierung der Symbole von Γ ist das immer möglich.

Codierung von M :

Codierung $\langle M \rangle$ einer TM $M = (Q, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\})$:

$$Q = \{q_1, q_2, q_3, \dots, q_n\}$$

$$\Gamma = \{x_1, x_2, x_3\}, \{L, R, N\} = \{d_1, d_2, d_3\}$$

$$\delta(q_i, x_j) = (q_k, x_l, d_m) \Rightarrow$$

$$\text{code}(q_i) = 0^i$$

$$\text{code}(x_i) = 0^i$$

$$\text{code}(d_i) = 0^i$$

$$\text{code}(\delta) = \text{code}(\delta(q_i, x_j) = (q_k, x_l, d_m)) = 0^i 1 0^j 1 0^k 1 0^l 1 0^m$$

zwei Übergänge trennen: 11, Anfang und Ende: 111

$$\Rightarrow \langle M \rangle = 111 \text{code}(\delta_1) 11 \text{code}(\delta_2) 11 \dots 11 \text{code}(\delta_p) 111$$

Diese Codierung ist nicht eineindeutig, da die $\text{code}(\delta_q)$ vertauscht werden können, aber ein Wort $x \in \{0, 1\}^*$ ist die Codierung höchstens einer TM M . Damit die Codierung eineindeutig wird, wird gefordert, dass die $\text{code}(\delta_q)$ in kanonisch aufsteigender Reihenfolge angeordnet werden (bzg. q, x).

Die durch ein Wort $x \in \{0, 1\}^*$ codierte TM soll als TM M_x bezeichnet werden.

Wir definieren ferner, dass für alle x , die nicht Codierung einer normierten TM sind, $M_x = M_0$ sein soll, wobei M_0 eine TM sei, die die leere Sprache akzeptiert.

Beispiel 9.1:

gegeben: 111 01010010100 11010010100100 111

111 010 1 0010100 11 0100 1 0100100 111

$$\Rightarrow \delta(q_1, x_1) = (q_2, x_1, d_2), \quad \delta(q_1, x_2) = (q_1, x_2, d_2)$$

$$\Rightarrow \delta(q_1, 0) = (q_2, 0, R), \quad \delta(q_1, 1) = (q_1, 1, R)$$

\Rightarrow sinnvolle Codierung.

Eingeführt wird jetzt eine

9.3.2 Universelle TM

Def. 9.5: Eine TM $U = (Q, \Sigma, \Gamma, \delta, q_1, B, F)$ heißt universell, wenn sie bei Eingabe $\langle M \rangle x$ das Ein-/Ausgabeverhalten von M auf x simuliert, d.h. wenn für alle TM-Codes $\langle M \rangle \in L_{TM-Code}$ und alle $x \in \{0, 1\}^*$ gilt:

- U hält auf $\langle M \rangle x \Leftrightarrow M$ hält auf x .

- U akzeptiert $\langle M \rangle x \Leftrightarrow M$ akzeptiert x .

- U verwirft $\langle M \rangle x \Leftrightarrow M$ verwirft x .
- $f_U(\langle M \rangle x) = f_M(x)$.
- Auf Eingaben, die nicht die Form $\langle M \rangle x$ haben, hält U nicht.

9.3.3 Eine nicht rekursiv aufzählbare Sprache

Wir sortieren $\{0, 1\}^*$ in kanonischer Ordnung.

$x_j \in \{0, 1\}^*$ sei das j -te Wort und M_i die i -te TM, deren Codierung die ganze Binärzahl i ist.

	\dots	x_l	x_{l+1}	x_{l+2}	x_{l+3}	x_{l+4}	x_{l+5}	\dots	x_j	\dots
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\dots		
M_l	\dots	1	0	0	1	1	0	\dots		
M_{l+1}	\dots	0	0	0	1	1	0	\dots		
M_{l+2}	\dots	1	1	1	0	0	\dots			
M_{l+3}	\dots	1	1	0	0	1	\dots			
M_{l+4}	\dots	0	1	1	1	0	\dots			
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\dots		
M_i	\dots	$M_{i,j}$	$= \begin{cases} 0 & \text{falls } x_j \notin L(M_i) \\ 1 & \text{falls } x_j \in L(M_i) \end{cases}$							
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\dots		

Wir konstruieren eine Sprache L_d , die sich auf die Einträge der Diagonale bezieht.

L_d soll eine Sprache sein, die von keiner TM erkannt wird:

$x_i \in L_d \Leftrightarrow (i, i)$ -Eintrag = 0, d.h. x_i wird nicht von M_i akzeptiert.

Annahme, L_d wird von einer TM M_j akzeptiert, dh. $L_d = L(M_j)$.

Wir erhalten folgenden Widerspruch:

- $x_j \in L_d \Rightarrow x_j \in L(M_j)$, da $L_d = L(M_j)$ nach Voraussetzung.
- $x_j \in L_d \Rightarrow (j, j)$ -Eintrag = 0 $\Rightarrow x_j \notin L(M_j)$ laut Definition.

Da x_j sowohl in $L(M_j) = L_d$ liegt als auch nicht in $L(M_j) = L_d$ liegt, nehmen wir an, dass unsere Voraussetzung $L_d = L(M_j)$ falsch ist, dh. es gibt in unserer Liste keine TM, die L_d akzeptiert und folglich überhaupt keine TM, die L_d akzeptiert.

Damit ist L_d nicht rekursiv aufzählbar (und schon gar nicht rekursiv).

9.3.4 Unentscheidbarkeit des Halteproblems

Def. 9.6: Das spezielle Halteproblem ist die Sprache

$$K = \{\langle M \rangle \mid M \text{ normierte TM und } M \text{ hält auf } \langle M \rangle\}.$$

K heißt auch Selbstanwendungssprache oder Diagonalsprache.

Satz 9.8: Das spezielle Halteproblem K ist nicht entscheidbar.

Es gilt: (i) K ist nicht rekursiv, (ii) \overline{K} ist nicht rekursiv aufzählbar.

Beweis: Es gilt $K = \overline{L_d}$.

Mittels universeller (3-Bd.) TM kann gezeigt werden, dass es eine TM M_i gibt, die $\langle M_i \rangle$ akzeptiert, dh. K ist rekursiv aufzählbar.

Da aber L_d nicht rekursiv aufzählbar ist, kann K nicht rekursiv sein und damit ist K nicht entscheidbar.

Man kann oft die Entscheidbarkeit eines Problems durch Zurückführung (Reduktion) auf ein anderes Problem, dessen Entscheidbarkeit bereits feststeht, beweisen.

Die Reduktion eines Problems bedeutet die Einbettung desselben als Spezialfall des anderen.

Def. 9.7: Seien $L_1, L_2 \subseteq \Sigma^*$ Sprachen. Dann heißt L_1 auf L_2 reduzierbar, symbolisch: $L_1 \leq L_2$, wenn es eine totale und berechenbare Funktion

$f : \Sigma^* \rightarrow \Sigma^*$ gibt, so dass gilt: $\forall x \in \Sigma^*$ gilt $x \in L_1 \Leftrightarrow f(x) \in L_2$.

Lemma: Falls $L_1 \leq L_2$ und L_2 rekursiv (bzw. rekursiv aufzählbar) ist, so ist auch L_1 rekursiv (bzw. rekursiv aufzählbar).

Folgerung aus dem Lemma:

Falls $L_1 \leq L_2$ und L_1 nicht rekursiv (bzw. nicht rekursiv aufzählbar) ist, so ist auch L_2 nicht rekursiv (bzw. nicht rekursiv aufzählbar).

Def. 9.8: Das allgemeine Halteproblem ist die Sprache

$$H = \{\langle M \rangle \# x \mid \text{TM } M \text{ hält auf } x\}.$$

Satz 9.9: Das Halteproblem H ist nicht entscheidbar (aber halbentscheidbar).

Beweis:

1. H rekursiv aufzählbar: Entwurf einer TM M , die H akzeptiert.
 M arbeitet auf Eingabe $y \in \{0, 1\}^*$ wie folgt:
 Teste, ob $y = \langle M \rangle \# x$. Falls NEIN Endlosschleife.
 Sonst starte U auf y . Falls U hält, akzeptiere.

2. \overline{H} nicht rekursiv aufzählbar \Rightarrow mit 1.) H nicht rekursiv:

Gemäß der Folgerung aus dem Lemma reicht es, $K \leq H$ nachzuweisen.

D.h. für $f : \Sigma^* \rightarrow \Sigma^*$ gilt: $\forall x \in \Sigma^* : x \in K \Leftrightarrow f(x) \in H$. Wähle daher

$$f(x) = \begin{cases} x\#x & \text{falls } x = \langle M \rangle \text{ für eine TM } M; \\ \varepsilon & \text{sonst.} \end{cases}$$

f ist total rekursiv. Für jedes $x \in \{0, 1\}^*$ gilt: $x \in K \Rightarrow$

es gibt eine TM M mit $x = \langle M \rangle$ und M hält auf $x \Rightarrow f(x) = x\#x \in H$.

Da $\varepsilon \notin H$ ist gilt umgekehrt: $f(x) \in H \Rightarrow f(x) = x\#x$, es gibt eine TM M mit $x = \langle M \rangle$ und M hält auf $x \Rightarrow x \in K$.

Somit gilt $K \leq H$ und mit gleichem f gilt $\overline{K} \leq \overline{H}$.

Daraus folgt: H nicht rekursiv und \overline{H} nicht rekursiv aufzählbar.

Um zu zeigen, wie mit der Reduktionsmethode gearbeitet werden kann,

Beispiel 9.2: Zeigen Sie:

- (a) $L = \{\langle M \rangle \mid M \text{ hält auf Eingabe } 0\}$ ist rekursiv aufzählbar,
- (b) \overline{L} ist nicht rekursiv aufzählbar. (Daraus folgt L nicht rekursiv.)

zu (a) Entwurf einer TM M_0 , die L akzeptiert: Die TM M_0 arbeitet auf Eingabe $x \in \{0, 1\}^*$ wie folgt:

1. Teste, ob x eine TM-Kodierung $\langle M \rangle$ ist. Falls nein, gehe in Endlosschleife.
2. Sonst starte die universelle TM U auf Eingabe $\langle M \rangle 0$
 Falls U hält, akzeptiere

Es gilt: M_0 akzeptiert die Eingabe $x \in \{0, 1\}^* \Rightarrow x$ ist eine TM-Kodierung $\langle M \rangle$ für TM M und U hält auf $\langle M \rangle 0 \Rightarrow M$ hält auf $0 \Rightarrow x \in L$.

Umgekehrt gilt: $x \in L \Rightarrow x$ ist eine TM-Kodierung $\langle M \rangle$ für TM M und U hält auf $\langle M \rangle 0 \Rightarrow M_0$ akzeptiert die Eingabe $x \in \{0, 1\}^*$.

zu (b) Zeigen, dass $\overline{H} \leq \overline{L}$ gilt, oder besser $H \leq L$:

Finde eine rekursive Funktion $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ mit

$$\forall x \in \{0, 1\}^* : x \in H \Leftrightarrow f(x) \in L.$$

Interessant sind nur Werte $f(x)$, die TM-Kodierungen sind.

Zu jedem x muss eine TM M_x und ihre Kodierung $f(x) = \langle M_x \rangle$ angegeben werden, so dass

$$\forall x \in \Sigma^* : x \in H \Leftrightarrow \langle M_x \rangle \in L.$$

oder

$$x \in H \Leftrightarrow M_x \text{ h\u00e4lt auf Eingabe } 0.$$

Idee: M_x h\u00e4lt auf allen Eingaben, also auch auf 0, wenn $x \in H$.

M_x arbeitet auf Eingabe $y \in \{0, 1\}^*$ wie folgt:

1. \u00dcberschreibe y mit x (x in der Steuereinheit von M_x gespeichert)
2. Starte die universelle TM U auf Eingabe x

Gezeigt werden muss

1. $f(x)$ ist rekursiv: Aus M_x kann $\langle M_x \rangle$ algorithmisch erzeugt werden.
2. F\u00fcr $x \in \{0, 1\}^*$ gilt nach der Definition von $f(x) = \langle M_x \rangle \in L$:
 $x \in H \Rightarrow U$ h\u00e4lt auf $x \Rightarrow M_x$ h\u00e4lt auf allen $y \Rightarrow f(x) = \langle M_x \rangle \in L$
 und $f(x) = \langle M_x \rangle \in L \Rightarrow M_x$ h\u00e4lt auf 0 $\Rightarrow U$ h\u00e4lt auf $x \Rightarrow x \in H$

Damit gilt $H \leq L$ mittels f .

Beispiel 9.3: Zeigen Sie:

- (a) $L = \{\langle M \rangle \mid M \text{ h\u00e4lt auf allen Eingabe } x \in \Sigma^*\}$ ist nicht rekursiv aufz\u00e4hlbar, dh. $\overline{H} \leq L$ bzw. $H \leq \overline{L}$.
- (b) \overline{L} ist nicht rekursiv aufz\u00e4hlbar, dh. $\overline{H} \leq \overline{L}$ bzw. $H \leq L$ wie im vorhergehenden Beispiel.
- (a) Finde eine rekursive Funktion $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ mit

$$\forall x \in \{0, 1\}^* : x \in H \Leftrightarrow f(x) = \langle M_x \rangle \in \overline{L}.$$

$f(x)$ soll folgende Eigenschaft besitzen:

$$\forall x \in \{0, 1\}^* := \begin{cases} x \in H \Rightarrow H_{M_x} \text{ ist endlich;} \\ x \notin H \Rightarrow H_{M_x} = \{0, 1\}^*. \end{cases}$$

M_x arbeitet auf Eingabe $y \in \{0, 1\}^*$ wie folgt:

1. Schreibe x auf Bd.2 (x in der Steuereinheit von M_x gespeichert).
2. Lasse die universelle TM U auf x für $|y|$ Schritte laufen.
3. Falls U hält, gehe in eine Endlosschleife (Berechnung hält, genau dann wenn $|y| < t_U(x)$), sonst halte.

M_x und $\langle M_x \rangle$ können aus x bestimmt werden und damit ist f rekursiv.

Außerdem gilt: $\forall x \in \{0, 1\}^* := \begin{cases} x \in H & \Rightarrow H_{M_x} = \{y \mid |y| < t_U(x)\} \\ x \notin H & \Rightarrow H_{M_x} = \{0, 1\}^*. \end{cases}$

Damit gilt für alle x : $x \notin H \Leftrightarrow H_{M_x} = \{0, 1\}^* \Leftrightarrow f(x) \in L$.

Def. 9.9: Die Haltemenge für eine TM M ist die Sprache

$$H_M = \{x \in \Sigma^* \mid M \text{ hält auf } x\}.$$

Satz 9.10: Die Haltemenge H_M ist nicht entscheidbar.

Beweisidee: Mittels Satz von RICE zeigt man, dass H_M nicht rekursiv ist und mittels Entwurf einer TM M' zeigt man, dass H_M rek. aufzählbar ist.

9.3.5 Der Satz von RICE

Def. 9.10: Eine Teilmenge \mathcal{E} der Menge aller r.a. Sprachen über $\{0, 1\}$,

$\mathcal{E} \subseteq \{L \subseteq \{0, 1\}^* \mid L \text{ r.a.}\}$, nennt man Eigenschaft r.a. Sprachen.

\mathcal{E} heißt triviale Eigenschaft, falls $\mathcal{E} = \emptyset$ oder $\mathcal{E} = \{L \subseteq \{0, 1\}^* \mid L \text{ r.a.}\}$.

z.B.: - Endlichkeit: $\mathcal{E} = \{L \subseteq \{0, 1\}^* \mid L \text{ r.a., } L \text{ endlich}\}$

- Rekursivität: $\mathcal{E} = \{L \subseteq \{0, 1\}^* \mid L \text{ r.a., } L \text{ rekursiv}\}$

Eine Sprache L hat die Eigenschaft \mathcal{E} , wenn $L \in \mathcal{E}$.

Def. 9.11: Eine Teilmenge \mathcal{F} der Menge aller partiell rekursiven Funktionen über $\{0, 1\}$, $\mathcal{F} \subseteq \{f : \{0, 1\}^* \rightarrow \{0, 1\}^* \mid f \text{ partiell rekursiv}\}$, nennt man Eigenschaft partiell rekursiver Funktionen. \mathcal{F} heißt triviale Eigenschaft, falls $\mathcal{F} = \{f : \{0, 1\}^* \rightarrow \{0, 1\}^* \mid f \text{ partiell rekursiv}\}$ oder $\mathcal{F} = \emptyset$.

Der Satz von RICE besagt, dass für kein \mathcal{E} und kein \mathcal{F} anhand von $\langle M \rangle$ mit einer TM entschieden werden kann, ob H_M Eigenschaft \mathcal{E} oder \mathcal{F} hat, außer in völlig trivialen Fällen.

Satz 9.11: (Satz von RICE)

1. Es sei \mathcal{E} eine nichttriviale Eigenschaft von r.a. Sprachen, d.h.

$$\emptyset \neq \mathcal{E} \subsetneq \{L \subseteq \{0, 1\}^* \mid L \text{ r.a.}\}.$$

(D.h. es existieren r.a. Sprachen $L_0, L_1 \subseteq \{0, 1\}^*$ derart, dass L_1 die Eigenschaft \mathcal{E} hat und L_0 dagegen nicht.)

Dann gilt: $L_{\mathcal{E}} = \{\langle M \rangle \mid H_M \in \mathcal{E}\}$ ist nicht rekursiv.

2. Es sei \mathcal{F} eine nichttriviale Eigenschaft von partiell rekursiven Funktionen, d.h.

$$\emptyset \neq \mathcal{F} \subset \{f : \{0, 1\}^* \rightarrow \{0, 1\}^* \mid f \text{ partiell rekursiv}\}.$$

(D.h. es existieren partiell rekursive Funktionen $f_0, f_1 \subseteq \{0, 1\}^*$ derart, dass f_1 die Eigenschaft \mathcal{F} hat und f_0 dagegen nicht.)

Dann gilt: $L_{\mathcal{F}} = \{\langle M \rangle \mid f_M \in \mathcal{F}\}$ ist nicht rekursiv.

Beispiele: Im Folgenden sei $L \subseteq \{0, 1\}^*$ rekursiv aufzählbar.

Beispiel 9.4: $L = \{\langle M \rangle \mid M \text{ TM, } M \text{ hält auf keiner Eingabe}\}$

$$L = \{\langle M \rangle \mid M \text{ TM, } H_M = \emptyset\}$$

$$L_{\mathcal{E}} = \{\langle M \rangle \mid M \text{ TM, } H_M \in \mathcal{E}\} \text{ mit } \mathcal{E} := \{L' \subseteq \{0, 1\}^* \mid L' \text{ r.a., } L' = \emptyset\}$$

\mathcal{E} ist nichttriviale Eigenschaft, denn es gilt $\emptyset \in \mathcal{E}$ und $\{0, 1\}^* \notin \mathcal{E}$.

Nach Satz von RICE ist $L = L_{\mathcal{E}} = \{\langle M \rangle \mid M \text{ TM, } H_M = \emptyset\}$ nicht rekursiv.

Beispiel 9.5: $L = \{\langle M \rangle \mid M \text{ TM, } M \text{ hält auf } 101\}$

$$L = \{\langle M \rangle \mid M \text{ TM, } 101 \in H_M\}$$

$$L_{\mathcal{E}} := \{\langle M \rangle \mid M \text{ TM, } H_M \in \mathcal{E}\} \text{ mit } \mathcal{E} := \{L' \subseteq \{0, 1\}^* \mid L' \text{ r.a., } 101 \in L'\}$$

\mathcal{E} ist nichttriviale Eigenschaft, denn es gilt $\emptyset \notin \mathcal{E}$ und $\{0, 1\}^* \in \mathcal{E}$.

Nach Satz von RICE ist $L = \{\langle M \rangle \mid M \text{ TM, } 101 \in H_M\}$ nicht rekursiv.

Beispiel 9.6: $L = \{\langle M \rangle \mid M \text{ TM, } H_M \text{ ist r.a.}\}$.

$$L_{\mathcal{E}} := \{\langle M \rangle \mid M \text{ TM, } H_M \in \mathcal{E}\} \text{ mit } \mathcal{E} := \{L' \subseteq \{0, 1\}^* \mid L' \text{ r.a.}\}$$

\mathcal{E} ist triviale Eigenschaft, also keine Aussage nach Satz von RICE.

Da alle Haltemengen r.a. sind, ist $L = \{\langle M \rangle \mid M \text{ TM, } H_M \text{ ist r.a.}\}$ rekursiv.

Es muss nur geprüft werden, ob die Eingabe eine korrekte Darstellung der Codierung einer Turingmaschine ist. Das ist über eine Syntaxprüfung immer entscheidbar.

Beispiel 9.7: $L = \{\langle M \rangle \mid M \text{ TM, } H_M \text{ ist rekursiv}\}$

$L_{\mathcal{E}} := \{\langle M \rangle \mid M \text{ TM, } H_M \in \mathcal{E}\}$ mit $\mathcal{E} := \{L' \subseteq \{0, 1\}^* \mid L' \text{ r.a., } L' \text{ rekursiv}\}$

\mathcal{E} ist nichttriviale Eigenschaft, denn es gilt $\emptyset \in \mathcal{E}$ und $H \notin \mathcal{E}$.

Nach Satz von RICE ist $L = \{\langle M \rangle \mid M \text{ TM, } H_M \text{ ist rekursiv}$ nicht rekursiv.

Kapitel 10

Komplexität - eine Einführung

10.1 Komplexitätsbegriff

Sprachen (und damit auch Funktionen, Probleme, Algorithmen etc.) können nach dem Bedarf an Zeit, Speicherplatz, Zuständen o.ä. Ressourcen zu ihrer Erkennung mittels Turingmaschine klassifiziert werden.

Diese Klassifizierungen bewerten die Kompliziertheit oder Komplexität der Berechnungen als Funktion der Eingabelänge $|x| = n$.

Komplexitätsmaße können abstrakt definiert und viele Ergebnisse allgemein bewiesen werden.

Die wichtigsten Berechnungskomplexitäten sind Zeitkomplexität und Platzkomplexität.

Def. 10.1: Sei M eine k -Bd.-TM und $x \in \Sigma^*$ für das Eingabealphabet Σ von M . Wir definieren

$$t_M(x) = \begin{cases} \text{Anzahl der Schritte, die } M \text{ auf } x \text{ ausführt} \\ (\in \mathbb{N} \cup \{\infty\}, t_M(x) = i \text{ für } k_0 \stackrel{i}{\vdash}_M k \text{ mit } k_0 \text{ Initialkonfiguration} \\ \text{und } k \text{ Haltekonfiguration und } t_M(x) = \infty \text{ falls } M \text{ auf } x \text{ nicht hält.}) \end{cases}$$
$$s_M(x) = \begin{cases} \text{Anzahl der verschiedenen Zellen auf allen Bändern,} \\ \text{auf die } M \text{ auf Eingabe } x \text{ schreibend zugreift (kann auch } \infty \text{ sein).} \end{cases}$$

Im allgemeinen sucht man möglichst effiziente Algorithmen zur Lösung eines Problems und deren Komplexität.

Hierbei interessiert in der Regel der „worst case“-Aufwand.

Für zusammengefasste Eingaben der Länge n wird definiert:

Def. 10.2: Sei M eine k -Bd.-TM und $x \in \Sigma^*$ für das Eingabealphabet Σ von M . Wir definieren

$$T_M(n) = \begin{cases} \max \{ t_M(x) \mid x \in \Sigma^n \} \\ \text{(Zeitbedarf im schlechtesten Fall über alle Eingaben} \\ \text{der Länge } n : \text{worst-case-Zeitkomplexität)} \end{cases}$$

$$S_M(n) = \max \{ s_M(x) \mid x \in \Sigma^n \}$$

Def. 10.3: Sei $t : \mathbb{N} \rightarrow \mathbb{R}_+$ eine Funktion.

Eine Turing-Maschine M heißt $t(n)$ -zeitbeschränkt, falls $T_M(n) \leq t(n)$ für alle $n \in \mathbb{N}$ gilt, dh. falls $t_M(x) \leq t(|x|)$ für alle $x \in \Sigma^*$ gilt.

Man kann eine Turing-Maschine M z.B. als n^3 -zeitbeschränkt bezeichnen, wenn $T_M(n) \leq n^3$ für alle $n \in \mathbb{N}$ gilt.

Def. 10.4: Sei $s : \mathbb{N} \rightarrow \mathbb{R}_+$ eine Funktion.

Eine Turing-Maschine M heißt $s(n)$ -platzbeschränkt, falls $S_M(n) \leq s(n)$ für alle $n \in \mathbb{N}$ gilt, dh. falls $s_M(x) \leq s(|x|)$ für alle $x \in \Sigma^*$ gilt.

In vielen Fällen lässt sich eine einfache obere Schranke $t(n)$ bzw. $s(n)$ im Gegensatz zu den Funktionen $T_M(n)$ bzw. $S_M(n)$ angeben, die oft einen unübersichtlichen Verlauf haben können.

Man beachte, wenn „ M $t(n)$ -zeitbeschränkt“ ist und es gilt $t(n) \leq t'(n)$, dann ist auch „ M $t'(n)$ -zeitbeschränkt“.

Für Funktionen $f, g : \mathbb{N} \rightarrow \mathbb{R}_+$ schreiben wir $g \in O(f)$ oder $g = O(f)$, wenn es eine Konstante $c > 0$ und ein $n_0 \in \mathbb{N}$ gibt, so dass gilt: $\forall n \geq n_0 : g(n) \leq c \cdot f(n)$.

Aus den Definitionen heraus kann man sich leicht überlegen, dass für jede k -Band-Turingmaschine M gilt:

Wenn M $t(n)$ -zeitbeschränkt ist, dann ist M $k \cdot t(n)$ -platzbeschränkt.

Aus praktischen Gründen wird oft für Komplexitätsuntersuchungen eine Turingmaschine mit k beidseitig unendlichen Bändern genutzt, von denen das erste die Eingabe enthält und das k -te Band die Ausgabe. Die Turingmaschine wird als k -Bd.-Turingmaschine mit Ein-/Ausgabe bezeichnet.

Obwohl die k -Band-Turingmaschinen zur 1-Band-Turingmaschine bezüglich der Berechenbarkeit äquivalent sind, hat die Bandanzahl Einfluss auf die Zeitkomplexität. Eine $f(n)$ -zeitbeschränkte Mehrband-Turingmaschine kann durch eine $O(f^2(n))$ -zeitbeschränkte 1-Band-Turingmaschine simuliert werden. Wir erhalten mit k -Band-Turingmaschinen realistischere Maße.

10.2 Komplexitätsklassen

10.2.1 Komplexitätsklassen

Seien $t, s : \mathbb{N} \rightarrow \mathbb{R}_+$ Funktionen.

Def. 10.5:

Eine nichtdeterministische Turing-Maschine N heißt $t(n)$ -zeitbeschränkt, wenn für jedes $x \in \Sigma^*$ die längste Berechnung von N auf x maximal $t(|x|)$ Schritte macht.

Def. 10.6:

Eine nichtdeterministische Turing-Maschine N heißt $s(n)$ -platzbeschränkt, wenn sie für jedes $x \in \Sigma^*$ für die längste Berechnung von N auf x auf maximal $s(|x|)$ Zellen schreibend zugreift.

In Anbetracht der Kürze der Zeit wollen wir hier hauptsächlich die Zeitkomplexität betrachten.

Def. 10.7: Zeitkomplexitätsklassen:

$$\text{DTIME}(t(n)) := \{L_M \mid M \text{ ist deterministische k-Bd.-TM, } \exists c > 0 : M \text{ ist } c \cdot t(n)\text{-zeitbeschränkt}\}$$

$$\text{NTIME}(t(n)) := \{L_N \mid N \text{ ist nichtdeterministische k-Bd.-TM, } \exists c > 0 : N \text{ ist } c \cdot t(n)\text{-zeitbeschränkt}\}$$

$$\text{FDTIME}(t(n)) := \{f_M \mid M \text{ ist deterministische k-Bd.-TM, } \exists c > 0 : M \text{ ist } c \cdot t(n)\text{-zeitbeschränkt}\}$$

Def. 10.8: Klassen P, NP, und FP:

$$\begin{aligned} P &:= \bigcup \{ \text{DTIME}(n^j) \mid j \geq 1 \} \\ &= \{L_M \mid \exists c > 0, j \geq 1 : M \text{ ist } c \cdot n^j\text{-zeitbeschränkte deterministische k-Bd.-TM}\} \end{aligned}$$

$$\begin{aligned} \text{NP} &:= \bigcup \{ \text{NTIME}(n^j) \mid j \geq 1 \} \\ &= \{L_N \mid \exists c > 0, j \geq 1 : M \text{ ist } c \cdot n^j\text{-zeitbeschränkte nichtdeterministische k-Bd.-TM}\} \end{aligned}$$

$$\begin{aligned} \text{FP} &:= \bigcup \{ \text{FDTIME}(n^j + 1) \mid j \geq 1 \} \\ &= \{f_M \mid \exists c > 0, j \geq 1 : M \text{ ist } c \cdot (n^j + 1)\text{-zeitbeschränkte deterministische k-Bd.-TM}\} \end{aligned}$$

Jede Sprache in P und jede Funktion in FP ist rekursiv, da die TM, die eine rekursive Sprache entscheidet und die TM, die eine rekursive Funktion berechnet, auf allen Eingaben hält. Aber nicht jede rekursive Sprache liegt in P und nicht jede rekursive Funktion liegt in FP, da nicht jede deterministische TM in polynomieller Zeit arbeitet.

10.2.2 Beispiele für Sprachen aus P und Funktionen aus FP

Für die Beschreibung der Beispiele brauchen wir eine Kodierung von Graphen über einem endlichen Alphabet.

Es sei $G = (V, E)$ ein (gerichteter oder ungerichteter) Graph mit $V = \{1, \dots, m\}$ und $E = \{(v_1, w_1), \dots, (v_\ell, w_\ell)\}$ mit $v_1, \dots, v_\ell, w_1, \dots, w_\ell \in V$ (geordnete oder ungeordnete Paare).

Adjazenzmatrixdarstellung:

Matrix $A_G = (a_{ij})_{1 \leq i, j \leq m}$ mit $(a_{ij}) = 1$ falls $(i, j) \in E$ und $(a_{ij}) = 0$ sonst.

$G = (V, E)$ wird dann kodiert als $\langle G \rangle := 0^m 1 a_{11} \dots a_{1m} \dots a_{m1} \dots a_{mm}$.

Darstellung durch Kantenlisten:

$\langle\langle G \rangle\rangle := 0^m 1 0^\ell \text{bin}_l(v_1) \text{bin}_l(w_1) \dots \text{bin}_l(v_\ell) \text{bin}_l(w_\ell)$ mit $l := \lceil \log(m+1) \rceil$, dh. jede Binärdarstellung $\text{bin}(i)_l$ hat exakt l Bits.

Da gilt $|\langle\langle G \rangle\rangle| \leq |\langle G \rangle|^2$ und $|\langle G \rangle| = m + 1 + M^2 < (m + 1)^2 \leq |\langle\langle G \rangle\rangle|^2$, ist es egal, welche Kodierung genutzt wird wenn es um die Frage geht, ob eine Sprache von Graphkodierungen, die ein Graphproblem formalisiert, in P liegt oder nicht.

Beispiele:

1. $L_{\text{zush}} = \{\langle G \rangle \mid G \text{ ist zusammenhängender ungerichteter Graph}\}$
2. $L_{\text{stark-zush}} = \{\langle G \rangle \mid G \text{ ist stark zusammenhängender gerichteter Graph}\}$
(G heißt stark zusammenhängend, wenn für jedes Knotenpaar (u, v) in G ein gerichteter Weg von u nach v und von v nach u existiert.)
3. $L_{\text{Baum}} = \{\langle G \rangle \mid G \text{ ist zusammenhängend und kreisfrei}\}$
4. $f_{\text{Zush-Komp}} : \{0, 1\}^* \rightarrow \{0, 1\}^*$ mit der Eigenschaft:
 $f_{\text{Zush-Komp}}(\langle G \rangle) = \text{Liste der Zusammenhangskomponenten von } G;$
 $f_{\text{Zush-Komp}}(w) = \varepsilon, \text{ falls } w \text{ keinen ungerichteten Graphen kodiert.}$

10.3 NP-Probleme

Für die Praxis steht die Frage, ob es Probleme gibt, die zwar prinzipiell algorithmisch lösbar sind, aber keinen „effizienten“ Algorithmus besitzen. Effiziente Algorithmen werden im Bereich der Algorithmen mit polynomieller Laufzeit angesiedelt, sie können effizient sein, müssen aber nicht (z.B. falls der Exponent zu groß wird).

Um den Begriff „Polynomialzeitalgorithmus“ formulieren zu können, wird das Turingmaschinenmodell benutzt. Für die Eingaben genügt eine einfache Binärkodierung für Zahlen, Graphkodierungen und andere Strukturen.

Man kann sagen, dass eine Sprache $L \subseteq \Sigma^*$ (eine Funktion $f : \Sigma^* \rightarrow \Delta^*$) von einem Polynomialzeitalgorithmus genau dann berechnet werden kann, wenn es eine polynomiell zeitbeschränkte Turingmaschine M mit $L = L_M$ ($f = f_M$) gibt. (Siehe 10.2.2 $L \in P$ und $f \in FP$.)

Man kann schlussfolgern, dass eine Sprache L , für die es keine polynomiell zeitbeschränkte Turingmaschine gibt, durch keinen effizienten Algorithmus entscheidbar ist, bzw. dass eine Funktion f , für die es keine polynomiell zeitbeschränkte Turingmaschine gibt, durch keinen effizienten Algorithmus berechenbar ist.

Innerhalb von NP, der Klasse derjenigen Sprachen, die durch „Nichtdeterministische Polynomiell zeitbeschränkte TM'en“ entschieden werden können, findet man die „NP-vollständigen“ Sprachen, die als die „schwierigsten“ anzusehen sind. Für die NP-vollständigen Sprachen gibt es vermutlich keine polynomiell zeitbeschränkte Turingmaschinen, also auch keine effizienten Algorithmen.

Bezüglich der nichtdeterministischen polynomiell zeitbeschränkten TM'en gibt es noch die Klasse $\text{co-NP} := \{\bar{L} \mid L \in \text{NP}\}$. (Für $L \in P$ gilt $\bar{L} \in P$.)

Beispiel 10.1: „Ratewort-TM“, Nichtdeterministisches Schreiben eines Binärstrings.

Die Ratewort-TM M_{RW} arbeitet auf der Eingabe 0^n wie folgt:

Das Eingabewort 0^n wird (im Zs. q_0) von links nach rechts gelesen und dabei wird für jede 0 nichtdeterministisch 0 oder 1 geschrieben. Der LS-Kopf geht (im Zst. q_1) auf die Ausgangsposition zurück und hält (im Zst. q_2).

Die Ausgabe ist ein Binärwort $b_1 \dots b_n \in \{0, 1\}^n$. Es ist jedes solche Wort als Ausgabe möglich.

Formal ergibt sich für M_{RW} : $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0\}$, $\Gamma = \{0, 1, B\}$, $F = \{q_2\}$ und δ :

$$\delta(q_0, 0) = \{(q_0, 0, R), (q_0, 1, R)\},$$

$$\delta(q_0, B) = \{(q_1, B, L)\},$$

$$\delta(q_1, a) = \{(q_1, a, L)\}, \text{ für } a \in \{0, 1\}$$

$$\delta(q_1, B) = \{(q_2, B, R)\}.$$

Für alle restlichen Paare (q, a) gilt $\delta(q, a) = \emptyset$.

Auf die Eingabe 0^n macht M_{RW} $2(n+1)$ Schritte, sie ist also $2(n+1)$ -zeitbeschränkt.

Nach dem gleichen Prinzip kann man eine Rateword-TM für Σ entwerfen.

Beispiel 10.2: Cliquesproblem

1. Sei $G = (V, E)$ ein ungerichteter Graph. $V' \subseteq V$ heißt **Clique** in G , falls die Knoten V' einen vollständigen Teilgraphen bilden. Es gilt also: $\forall v, w \in V', v \neq w : (v, w) \in E$. $|V'|$ heißt Größe der Clique.

Z.B.

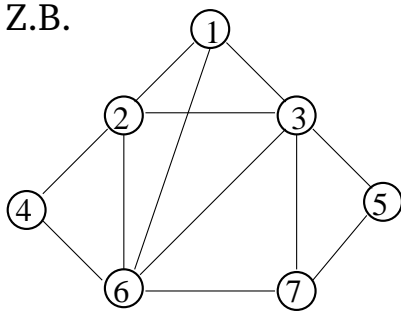


Bild 10.1: Graph mit einer maximalen Clique der Größe 4, $V' = \{1, 2, 3, 6\}$

2. Das **Cliquesproblem** ist die Aufgabe, in einem gegebenen Graphen eine möglichst große Clique zu finden. Wir betrachten von der Formulierung her drei verschiedene Varianten:

Variante 1: (Optimierungsproblem im eigentlichen Sinne, Suche nach einer optimalen Struktur)

Gegeben sei ein Graph $G = (V, E)$.

Aufgabe: Finde eine Clique $V' \subseteq V$, so dass $|V'| \geq |V''|$ für jede Clique $V'' \subseteq V$ in G .

Variante 2: (Parameteroptimierung)

Gegeben sei ein Graph $G = (V, E)$.

Aufgabe: Bestimme das maximale $k \in \mathbb{N}$, so dass G eine Clique der Größe k hat.

Variante 3: (Entscheidungsproblem)

Gegeben sei ein Graph $G = (V, E)$ und $k \in \mathbb{N}$.

Frage: Gibt es in G eine Clique V' der Größe $|V'| \geq k$?

Formal ergibt Variante 1 eine Funktion $f_{\text{Clique-Size}}$, die aus $\langle G \rangle$ die Binärzahl der Größe einer größten Clique berechnet und Variante 2 eine Funktion f , die aus $\langle G \rangle$ eine größte Clique V'_{max} berechnet. Darauf soll hier nicht weiter eingegangen werden.

Variante 3 kann als Sprache über $\{0, 1\}$ angegeben werden:

$$L_{\text{Clique}} := \{ \langle G \rangle \text{ bin}(k) \mid k \geq 1, G \text{ Graph über } V = \{1, \dots, m\}, G \text{ hat Clique der Größe } k \}$$

Es gelten folgende Implikationen:

Wenn $f \in \text{FP}$, dann auch $f_{\text{Clique-Size}} \in \text{FP}$, wenn $f_{\text{Clique-Size}} \in \text{FP}$, dann auch $L_{\text{Clique}} \in \text{P}$ und wenn $L_{\text{Clique}} \in \text{P}$, dann auch $f \in \text{FP}$. D.h. wenn $L_{\text{Clique}} \in \text{P}$, dann haben alle 3 Varianten einen Polynomialzeitalgorithmus.

Wir werden sehen, dass die Vermutung nahe liegt, dass L_{Clique} nicht in P liegt.

Wir zeigen dazu, dass L_{Clique} in NP liegt:

Angegeben wird eine nichtdeterministische, polynomiell zeitbeschränkte NTM N mit $L_N = L_{\text{Clique}}$, die auf Eingabe $w \in \{0, 1\}^*$ wie folgt arbeitet:

0. Hat w das Format $\langle G \rangle \text{ bin}(k)$ mit G Graph mit Knoten $V = \{1, \dots, m\}$ und $1 \geq k \geq m$. Falls **NEIN**, verwerfe. (determ.)
1. Ermittle m und schreibe 0^m auf Bd. 2. (determ.)
2. Lasse M_{RW} auf Bd. 2 laufen. **(nichtdeterm.)**
Auf Bd. 2 wird $b_1 \dots b_m \in \{0, 1\}^m$ erzeugt.
Die dadurch gelieferte Knotenmenge sei $V' = \{i \mid 1 \leq i \leq m, b_i = 1\}$.
3. Teste (determ.)
 - (α) $|b_1 \dots b_m|_1 \geq k$.
 - (β) V' ist Clique in G .
 Falls für α und β **JA**, halte akzeptierend, sonst halte verwerfend.

Zeit: 0. und 1. in $O(m^2)$, 2. in $O(2^m)$, 3. in $O((m^2)^l)$ für eine Konstante l , also polynomielle Zeit.

Sollte 2. deterministisch ausgeführt werden, müssten alle 2^m Möglichkeiten für V' erzeugt und getestet werden \Rightarrow exponentielle Zeit.

Beispiel 10.3: Rucksackproblem

1. Gegeben sind ein Rucksack mit Volumen b und m verformbare Gegenstände mit Volumina $a_1, \dots, a_m \in \mathbb{N}$ und Nutzenwerten $c_1, \dots, c_m \in \mathbb{N}$
2. Das Rucksackproblem ist die Aufgabe, einige der Gegenstände in den Rucksack zu packen und dabei den Gesamtnutzen zu maximieren, ohne die Volumenschranke zu überschreiten. Es können 3 Varianten wie folgt angegeben werden:

Variante 1: (Optimierungsproblem im eigentlichen Sinne, Suche nach einer optimalen Struktur)

Gegeben seien $b, a_1, \dots, a_m, c_1, \dots, c_m \in \mathbb{N}$.

Aufgabe: Finde $I \subseteq \{1, \dots, m\}$ mit $\sum_{i \in I} a_i \leq b$, derart dass gilt:

$$\forall J \subseteq \{1, \dots, m\} : \sum_{i \in J} a_i \leq b \Rightarrow \sum_{i \in J} c_i \leq \sum_{i \in I} c_i$$

Variante 2: (Parameteroptimierung)

Gegeben Gegeben seien $b, a_1, \dots, a_m, c_1, \dots, c_m \in \mathbb{N}$.

Aufgabe: Bestimme $k = \max \left\{ \sum_{i \in I} c_i \mid I \subseteq \{1, \dots, m\}, \sum_{i \in I} a_i \leq b \right\}$

Variante 3: (Entscheidungsproblem)

Gegeben seien $b, a_1, \dots, a_m, c_1, \dots, c_m \in \mathbb{N}$ und $k \in \mathbb{N}$.

Frage: Gibt es $I \subseteq \{1, \dots, m\}$ mit $\sum_{i \in I} a_i \leq b$ und $\sum_{i \in I} c_i \geq k$?

Variante 3 als Sprache über $\{0, 1, \#\}$ formuliert:

$$L_{RS} = \{ \text{bin}(a_1) \# \dots \# \text{bin}(a_m) \#\# \text{bin}(c_1) \# \dots \# \text{bin}(c_m) \# \text{bin}(b) \# \text{bin}(k) \mid \\ m \geq 1, a_1, \dots, a_m, c_1, \dots, c_m \in \mathbb{N}, b, k \in \mathbb{N}, \\ \exists I \subseteq \{1, \dots, m\} : \sum_{i \in I} a_i \leq b \wedge \sum_{i \in I} c_i \geq k \}$$

Es kann wieder eine NTM N entworfen werden, die L_{RS} entscheidet.

Beachte: Eine NTM N_L akzeptiert $w \in \Sigma^*$, wenn das Eingabewort w zur Sprache L gehört **und** die NTM richtig gewählt hat.

Gilt $w \in L$ und die NTM N_L hat nicht richtig gewählt, dann verwirft die NTM N_L , d.h. Verwerfen hat keinen Aussagewert.

Entscheidungsvariante als Sprache:

$$L_{IS} := \{(G, k) \mid G = (V, E) \text{ Graph}, k \geq 1; \text{ es gibt } V' \subseteq V \text{ mit } |V'| \geq k \text{ und } \forall (v, w) \in V' : (v, w) \notin E\}$$

Zwischen L_{IS} , L_{VC} und L_{Clique} gibt es \leq_p -Reduktionen.

Satz 10.1: (a) $L_{Clique} \leq_p L_{IS}$ und $L_{IS} \leq_p L_{Clique}$

(b) $L_{IS} \leq_p L_{VC}$ und $L_{VC} \leq_p L_{IS}$

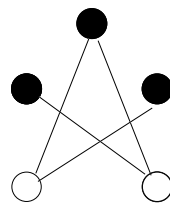
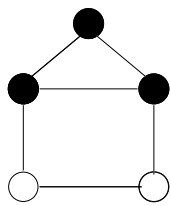


Bild 10.4: schwarz: Clique in G , schwarz: \bar{G} unabhängige Menge in \bar{G} ,
weiß: Knotenüberdeckung in \bar{G}

zu (a) Zu $G = (V, E)$ ist $\bar{G} = (V, \bar{E})$ mit $(v, w) \in \bar{E} \Leftrightarrow (v, w) \notin E$ für $v, w \in V$.

$L_{Clique} \leq_p L_{IS}$ mittels f mit $f((G, k)) := (\bar{G}, k)$. (Beweis weglassen ?)

Hilfsbehauptung 1:

Für $V' \subseteq V$ gilt: V' Clique in $(V, E) \Leftrightarrow V'$ unabhängig in (V, \bar{E})

$$\begin{aligned} (G, k) \in L_{Clique} &\stackrel{\text{Def.}}{\Leftrightarrow} G \text{ hat Clique der Größe } k \\ &\stackrel{\text{HB1}}{\Leftrightarrow} \bar{G} \text{ hat unabhängige Menge der Größe } k \\ &\stackrel{\text{Def.}}{\Leftrightarrow} f((G, k)) := (\bar{G}, k) \in L_{IS} \end{aligned}$$

Genauso $L_{IS} \leq_p L_{Clique}$ mittels f mit $f((G, k)) := (\bar{G}, k)$.

zu (b) $L_{IS} \leq_p L_{VC}$ mittels g mit $g(((V, E), k)) := ((V, E), |V| - k)$

Hilfsbehauptung 2: Für $V' \subseteq V$ gilt:

V' unabhängige Menge in $(V, E) \Leftrightarrow V - V'$ Knotenüberdeckung in (V, E) :

$$\begin{aligned} \exists v, w \in V' : (v, w) \in E &\Leftrightarrow \exists (v, w) \in E : v \in V' \wedge w \in V' \\ &\Leftrightarrow \exists (v, w) \in E : v \notin V - V' \wedge w \notin V - V'. \end{aligned}$$

$$\begin{aligned} ((V, E), k) \in L_{IS} &\text{ hat unabhängige Menge der Größe } \geq k \\ &\stackrel{\text{HB2}}{\Leftrightarrow} (V, E) \text{ hat Knotenüberdeckung der Größe } \leq |V| - k \\ &\Leftrightarrow g(((V, E), k)) = ((V, E), |V| - k) \in L_{VC}. \end{aligned}$$

Genauso $L_{VC} \leq_p L_{IS}$ mittels g mit $g(((V, E), k)) := ((V, E), |V| - k)$.

Es gibt Sprachen in NP, die besondere Eigenschaften bzgl. der Relation \leq_p haben, nämlich die NP-**vollständigen** Sprachen.

Def. 10.10:

(a) Eine Sprache heißt NP-**vollständig** (engl. NP-**complete**), falls

(i) $L \in \text{NP}$

(ii) für alle $L' \in \text{NP}$ gilt $L' \leq_p L$ (L ist NP-**schwer**, NP-**„hard“**, NP-**„hart“**)

(b) $\text{NPC} := \{L \mid L \text{ Sprache, } L \text{ NP-vollständig}\}$

Alle hier vorgestellten Sprachen sind in NPC.

Lemma 10.2: Für die Klasse P gilt

(a) $L \in P \wedge L' \leq_p L \Rightarrow L' \in P$.

(b) $L \in P \wedge \emptyset \neq L' \neq \Sigma^* \Rightarrow L \leq_p L'$.

Folgender Satz liefert eine zentrale strukturelle Aussage über die Klassen P, NP und NPC:

Satz 10.2: $\text{NPC} \cap P = \emptyset \Leftrightarrow P \neq \text{NP}$.

Beweis: der Äquivalenz der umgekehrten Aussage $\text{NPC} \cap P \neq \emptyset \Leftrightarrow P = \text{NP}$.

„ \Leftarrow “: Wenn $P = \text{NP}$ ist, dann betrachte irgendein $L \in P$, nur nicht \emptyset und nicht Σ^* . Es gilt

(i) $L \in P = \text{NP}$

(ii) $L \leq_p L'$ für alle $L' \in P = \text{NP}$ nach Lemma 10.2.

Also ist $L \in \text{NPC} \cap P$.

(D.h. alle Sprachen in P außer \emptyset und Σ^* sind NP-**vollständig**, falls $P = \text{NP}$.)

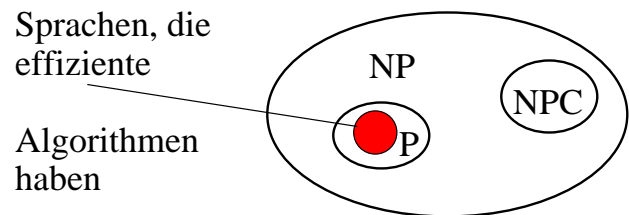
„ \Rightarrow “: Sei $L \in \text{NPC} \cap P$. Gezeigt werden soll $P = \text{NP}$. Weil $P \subseteq \text{NP}$ gilt, muss $\text{NP} \subseteq P$ gezeigt werden. Sei $L' \in \text{NP}$ beliebig. Aus $L \in \text{NPC}$ folgt nach Def.10.10, dass $L' \leq_p L$ gilt. Daraus und aus $L \in P$ folgt mit Lemma 10.2(a), dass $L' \in P$.

Die Frage $P \neq \text{NP}$? ist das berühmteste offene Problem der Komplexitätstheorie und wurde bereits 1970 formuliert.

Diese Frage ist äquivalent zu der für die Algorithmentheorie zentralen Frage „Ist es richtig, dass kein NP-vollständiges Problem einen Polynomialzeitalgorithmus, also einen Algorithmus mit vertretbarem Aufwand, hat?“

Man kann heute noch nicht sagen, ob die beiden Aussagen von Satz 10.2 zutreffen oder falsch sind. Angenommen wird aber, dass $P \neq NP$ ist. Als Indiz dafür, dass $NP \cap P = \emptyset$ gilt, spricht, dass alle NP-vollständigen Probleme Polynomialzeitalgorithmen besitzen würden, wenn $P = NP$ ist. Aber bis heute ist kein Problem gefunden worden.

Vermutet wird folgende Situation:



10.5 Der Satz von Cook

Im Zentrum der NP-Vollständigkeitstheorie steht eine spezielle Variante des Erfüllbarkeitsproblems.

Erfüllbarkeitsproblem: Gegeben ist eine KNF-Formel φ .

Finde eine erfüllende Belegung v für φ (KNF: Konjunktive Normalform).

Eine KNF-Formel $\varphi = C_1 \wedge \dots \wedge C_r$ heißt erfüllbar, falls es eine Belegung v gibt, so dass $v(\varphi) = 1$ (wahr) wird.

C_i heißt Klausel und ist eine Disjunktion von negierten und unnegierten Booleschen Variablen $X_j; \overline{X_j}$ (heißen Literale l_k).

Eine Belegung der Booleschen Variablen ist eine Abbildung

$$v : \{X_0, X_1, \dots\} \rightarrow \{0, 1\}$$

0 steht für den Wahrheitswert falsch und 1 für wahr.

Den Booleschen Variablen X_j werden damit Wahrheitswerte $v(X_j)$ und den entgegengesetzten Literalen $\overline{X_j}$ Wahrheitswerte $1 - v(X_j)$ zugeordnet.

So ergibt sich für Klauseln $v(C_i) \in \{0, 1\}$ und für KNF-Formeln $v(\varphi) \in \{0, 1\}$.

Das Erfüllbarkeitsproblem wird von einer Menge SAT charakterisiert:

$$SAT = \{\varphi \mid \varphi \text{ KNF-Formel, } \varphi \text{ erfüllbar}\}$$

SAT ist keine Sprache, da kein endliches Alphabet zugrunde liegt, sondern $\{X_0, X_1, \dots\} \cup \{\overline{X_0}, \overline{X_1}, \dots\} \cup \{(\, , \wedge, \vee\}$.

Mit entsprechender Kodierung kann man Formeln über dem Alphabet

$\Sigma = \{[,], (,), \wedge, \vee, \neg, 0, 1\}$ angeben. X_i kodiert man z.B. mit $[\text{bin}(i)]$ und $\overline{X_i}$ mit $\neg[\text{bin}(i)]$ für $i \in \mathbb{N}$.

Für $\varphi = (X_3 \vee X_5) \wedge (\overline{X_1} \vee X_4)$ ergibt sich $\langle \varphi \rangle = ([11] \vee [101]) \wedge (\neg[1] \vee [100])$

Damit folgt

Def. 10.11: Das Erfüllbarkeitsproblem für KNF-Formeln

$$L_{\text{SAT}} := \{\varphi \mid \varphi \text{ KNF-Formel, } \varphi \text{ erfüllbar}\} \text{ oder formal}$$

$$:= \{\langle \varphi \rangle \mid \varphi \text{ KNF-Formel, } \varphi \text{ erfüllbar}\}.$$

Aus technischen Gründen nutzt man auch das Erfüllbarkeitsproblem für 3-KNF-Formeln:

Def. 10.12: Das Erfüllbarkeitsproblem für 3-KNF-Formeln

$$L_{3\text{-SAT}} := \{\varphi \in L_{\text{SAT}} \mid \text{jede Klausel von } \varphi \text{ hat genau 3 Literale}\}$$

Das Erfüllbarkeitsproblem ist z.B. beim Entwurf von logischen Schaltungen relevant. Cook hat 1970 gezeigt, dass L_{SAT} NP-vollständig ist.

Satz 10.3: Satz von Cook L_{SAT} ist NP-vollständig.

Beweis: siehe Def. 10.10

(i) $L_{\text{SAT}} \in \text{NP}$:

Die NTM N arbeitet auf der Eingabe x wie folgt:

- 1.) Teste, ob $x = \langle \varphi \rangle$ für eine KNF-Formel φ . Falls **NEIN**, halte und verwerfe. DFA
- 2.) Schreibe auf Bd.2: $\#[\text{bin}(i_1)]\#[\text{bin}(i_2)]\#\dots\#[\text{bin}(i_m)]$, wobei $[\text{bin}(i_1)], [\text{bin}(i_2)], \dots, [\text{bin}(i_m)]$ genau die Kodierungen der in φ vorkommenden Variablen in beliebiger Reihenfolge aber ohne Wiederholung sind. Hierzu schreibt man die Variablen von Bd.1 ab und benutzt Textsuche, um Wiederholungen zu vermeiden.
- 3.) Schreibe 0^m auf Bd.3.
- 4.) Lasse M_{RW} auf Bd.3 laufen. Ergebnis $b_1 \dots b_m \in \{0, 1\}^*$.
- 5.) Ändere Bd.2 zu $b_1[\text{bin}(i_1)]b_2[\text{bin}(i_2)] \dots b_m[\text{bin}(i_m)]$
- 6.) Ersetze auf Bd.1 jedes Teilwort $[\text{bin}(i_r)]$ durch $b_r \dots b_r$ und jedes Teilwort $\neg[\text{bin}(i_r)]$ durch $\overline{b_r} \dots \overline{b_r}$. (Textsuche benutzen.)

- 7.) Teste, ob auf Bd.1 in jedem Paar von runden Klammern mindestens eine 1 vorkommt. Falls **Ja**, akzeptiere, falls **NEIN**, verwerfe. DFA
Man sieht, für jedes x gilt:

$$\begin{aligned}
 x \in L_{\text{SAT}} &\Leftrightarrow x = \langle \varphi \rangle \text{ für eine KNF-Formel } \varphi \\
 &\Leftrightarrow \text{es existiert eine Belegung } v \text{ (interessant nur die } m \text{ Werte} \\
 &\quad \text{für die in } \varphi \text{ vorkommenden Variablen) mit } v(\varphi) = 1 \\
 &\Leftrightarrow \text{es gibt ein Binärwort } b_1 \dots b_m, \text{ das in den Schritten 5.) bis} \\
 &\quad \text{7.) dazu führt, dass } N \text{ akzeptiert} \\
 &\Leftrightarrow \text{es gibt eine akzeptierende Berechnung für die in } x = \langle \varphi \rangle \\
 &\quad \text{vorkommenden Variablen.}
 \end{aligned}$$

N hat polynomielle Laufzeit.

- (ii) L_{SAT} NP-schwer: Der Teil des Beweises ist sehr viel schwieriger als der vorherige. Wer sich dafür interessiert, findet ihn unter /MD01/, S.165.

10.6 NP-vollständige Probleme

Damit man die NP-Vollständigkeit nicht immer wie im vorherigen Abschnitt beweisen muss, gibt es für die praktische Anwendung ein einfacheres Verfahren, nämlich die Reduktionsmethode.

Lemma 10.3: Reduktionsmethode

Eine Sprache L ist NP-vollständig, wenn gilt:

- (i) $L \in \text{NP}$.
- (ii)* $L' \leq_p L$ für ein $L' \in \text{NPC}$.

Gilt die Bedingung (ii)* \Leftrightarrow die Bedingung (ii) der Definition 10.10 gilt, d.h. $L'' \leq_p L$ für jedes $L'' \in \text{NPC}$?

Nach (ii)* ist $L' \in \text{NPC}$ und somit gilt $L'' \leq_p L'$ für jedes $L'' \in \text{NP}$.

Weiter ist nach (ii)* $L' \leq_p L$ und mit $L'' \leq_p L'$ gilt $L'' \leq_p L$ (Transitivität).

Man kann also folgendermaßen vorgehen, um zu zeigen dass L NP-vollständig ist:

- (i) Zeige, dass $L \in \text{NP}$ ist.
- (ii) Wähle eine geeignete als NP-vollständig bekannte Sprache L' und zeige (durch Konstruktion der Reduktionsfunktion), dass $L' \leq_p L$.

Beispiel 10.6: $L_{3\text{-SAT}}$ ist NP-vollständig:(i) Zeige, dass $L_{3\text{-SAT}} \in \text{NP}$ ist:

NTM N wie für L_{SAT} , nur dass der Syntaxcheck testet, ob $x = \langle \varphi \rangle$ für eine 3-KNF-Formel φ .

(ii) Wähle eine geeignete als NP-vollständig bekannte Sprache L' und zeige (durch Konstruktion der Reduktionsfunktion), dass $L' \leq_p L$.

Zeige $L_{\text{SAT}} \leq_p L_{3\text{-SAT}}$:

Wir müssen eine Reduktionsfunktion definieren, die eine KNF-Formel φ in eine 3-KNF-Formel φ^* transformiert derart, dass gilt:

$$\varphi \text{ ist erfüllbar} \Leftrightarrow \varphi^* \text{ ist erfüllbar.}$$

Wir betrachten nur KNF-Formeln φ , Eingaben, die nicht KNF-Formeln sind, werden auf nichterfüllbare 3-KNF-Formeln abgebildet.

Außerdem ignorieren wir die Kodierungen.

„ \Rightarrow “: φ ist erfüllbar $\Rightarrow \varphi^*$ ist erfüllbar.

$\varphi = C_1 \wedge \dots \wedge C_r$ sei eine KNF-Formel. Zu jeder Klausel C_j wird eine 3-KNF-Formel φ_j^* gebildet, so dass φ^* erzeugt werden kann:

$$\varphi^* = \varphi_1^* \wedge \dots \wedge \varphi_r^*$$

Die 3-KNF-Formel φ_j^* wird wie folgt aus $C_j = (l_1 \vee \dots \vee l_s)$ (j vorläufig fest) gebildet:

1. Fall $s = 1$: Wähle zwei neue Variable Z_1 und Z_2 , die sonst nirgends vorkommen (auch nicht in anderen Teilformeln $\varphi_{j'}^* \mid j' \neq j$) und definiere

$$\varphi_j^* = (l_1 \vee Z_1 \vee Z_2) \wedge (l_1 \vee Z_1 \vee \overline{Z_2}) \wedge (l_1 \vee \overline{Z_1} \vee Z_2) \wedge (l_1 \vee \overline{Z_1} \vee \overline{Z_2}).$$

2. Fall $s = 2$: Wähle eine neue Variable Z_1 und definiere

$$\varphi_j^* = (l_1 \vee l_2 \vee Z_1) \wedge (l_1 \vee l_2 \vee \overline{Z_1})$$

3. Fall $s = 3$: Definiere

$$\varphi_j^* = (l_1 \vee l_2 \vee l_3)$$

4. Fall $s \geq 4$: Wähle $s - 3$ neue Variable Z_3, Z_4, \dots, Z_{s-1} und definiere

$$\left. \begin{array}{l}
 \varphi_j^* = (l_1 \vee l_2 \vee Z_3) \\
 \wedge (\overline{Z_3} \vee l_3 \vee Z_4) \\
 \vdots \\
 \wedge (\overline{Z_{k-2}} \vee l_{k-2} \vee Z_{k-1})
 \end{array} \right\} v^*(Z_h) = 1$$

$$\wedge (\overline{Z_{k-1}} \vee l_k \vee Z_k)$$

$$\left. \begin{array}{l}
 \wedge (\overline{Z_k} \vee l_{k+1} \vee Z_{k+1}) \\
 \vdots \\
 \wedge (\overline{Z_{s-1}} \vee l_{s-1} \vee l_s)
 \end{array} \right\} v^*(Z_h) = 0, \text{ dh. } v^*(\overline{Z_h}) = 1$$

$$\varphi_j^* = C_{j1} \wedge C_{j2} \wedge \dots \wedge C_{j(s-2)}$$

$v(\varphi) = 1$, also auch $v(C) = 1$, also auch $v(C_j) = 1$, also auch $v(\varphi_j^*) = 1$. Damit gibt es ein k mit $v(l_k) = 1$.

Damit alle C_{j_i} den Wert 1 erhalten mit $1 \leq i \leq s - 2$, setze man

$$v^*(Z_h) = \begin{cases} 1, & \text{falls } h < k \\ 0, & \text{falls } h \geq k \end{cases}$$

So erhalten unter v^* alle Klauseln C_{j_i} mit $1 \leq i \leq s - 2$ den Wert 1:

- Da $v^*(l_k) = v(l_k) = 1$, hat die Klausel, die l_k enthält, unter v^* den Wert 1.
- Die vorhergehenden Klauseln enthalten eine Variable Z_h mit $v^*(Z_h) = 1$, die folgenden Klauseln enthalten ein Literal $\overline{Z_h}$ mit $v^*(\overline{Z_h}) = 1$.

Also gilt für alle Klauseln $v^*(C_{j_i}) = 1$ mit $1 \leq i \leq s - 2$, also φ^* ist erfüllbar.

„ \Leftarrow “ φ^* erfüllbar $\Rightarrow \varphi$ erfüllbar.

Für „ \Leftarrow “ gehe man indirekt vor.

Annahme: v^* nicht erfüllend für C_j . Man finde einen Widerspruch.

$(\overline{\varphi^* \Rightarrow \varphi} = \overline{\overline{\varphi^*} \vee \varphi} = \varphi^* \wedge \overline{\varphi})$: φ^* erfüllbar $\wedge \varphi$ nicht erfüllbar zum Widerspruch führen.

Annahme: $v^*(\varphi)$ nicht erfüllend für C_j , also φ nicht erfüllbar.

D.h. $\forall l_i : v(l_i) = v^*(l_i) = 0$ für $1 \leq i \leq s$.

Nach Konstruktion von φ^* gibt es kein k mit $v(l_k) = v^*(l_k) = 1$,
d.h. es gilt für alle Werte $v^*(Z_h) = 1$ für $3 \leq h \leq s - 1$,

aber damit ist $v^*((\overline{Z_{s-1}} \vee l_{s-1} \vee l_s)) = 0$.

Also wird auch $v^*(\varphi^*) = 0$, d.h. φ^* nicht erfüllbar.



\Rightarrow Annahme falsch.

Bemerkung:

$L_{\text{Clique}}, L_{\text{VC}}, L_{\text{IS}}, L_{\text{RS}}$ und viele andere Sprachen mehr sind ebenfalls in NPC.
(siehe Übung.)

Kapitel 11

Literatur

- /AST04/ ASTEROTH, ALEXANDER; BAIER, CHRISTEL:
Theoretische Informatik.
2002 by Pearson Studium.
- /SÖN92/ SCHÖNING, UWE:
Theoretische Informatik, kurz gefasst.
4. Aufl. Heidelberg, Berlin: Spektrum, Akad. Verlag 2001.
- /HOP94/ HOPCROFT, JOHN E.; MOTWANI, RAJEEV; ULLMAN, JEFFREY:
**Einführung in die Automatentheorie,
Formale Sprachen und Komplexitätstheorie.**
2. überarbeitete Aufl. 2002 by Pearson Studium.
- /POH93/ POHLERS, WOLFRAM:
Mathematische Grundlagen der Informatik.
München; Wien: Oldenbourg, 1993,
(Handbuch der Informatik, Bd. 1.5).
- /BRAU90/ BRAUER:
Grenzen maschineller Berechenbarkeit.
Informatik-Spektrum (1990) 13, S.61-70
Springer Verlag.
- /MD99/ DIETZFELBINGER, MARTIN:
Automaten und Formale Sprachen
Materialien zur Vorlesung, WS 2007,
TUI, Fak. IA, FG KTEA.
- /MD01/ DIETZFELBINGER, MARTIN:
Algorithmtheorie
Materialien zur Vorlesung, SS 2006,
TUI, Fak. IA, FG KTEA.